USENIX

conference

proceedings

9th USENIX Security Symposium

Denver, Colorado, USA
August 14–17, 2000

Sponsored by
USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Proceedings of the 9th USENIX Security Symposium

Denver, Colorado August 2000

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $27 for members and $35 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past USENIX Security Proceedings

Security VIII	August 1999	Washington, D.C., USA	$27/35
Security VII	January 1998	San Antonio, Texas, USA	$27/35
Security VI	July 1996	San Jose, California, USA	$27/35
Security V	June 1995	Salt Lake City, Utah, USA	$27/35
Security IV	October 1993	Santa Clara, California, USA	$15/20
Security III	September 1992	Baltimore, Maryland, USA	$30/39
Security II	August 1990	Portland, Oregon, USA	$13/16
Security	August 1988	Portland, Oregon, USA	$7/7

ISBN 1-880446-18-9

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the Ninth USENIX Security Symposium

August 14–17, 2000
Denver, Colorado, USA

Symposium Organizers

Program Co-Chairs

Steven Bellovin, *AT&T Labs—Research*
Greg Rose, *QUALCOMM Australia*

Invited Talks Coordinator

Win Treese, *Open Market Inc.*

Program Committee

Carl Ellison, *Intel Corporation*
Ian Goldberg, *University of California at Berkeley*
Peter Gutmann, *University of Auckland*
Trent Jaeger, *IBM T.J. Watson Research Center*
Markus Kuhn, *University of Cambridge, U.K.*
Marcus Leech, *Nortel*
Alain Mayer, *Bitmo*
Avi Rubin, *AT&T Labs—Research*
Jeff Schiller, *MIT*
Jonathan Trostle, *Cisco*
Wietse Venema, *IBM T.J.Watson Research Center*
Dan Wallach, *Rice University*
Tara Whalen, *Communications Research Centre Canada*
Elizabeth Zwicky, *Counterpane Internet Security*

External Reviewers

Matt Blaze, *AT&T Labs—Research*
John Ioannidis, *AT&T Labs—Research*
Trevor Jim, *AT&T Labs—Research*
Paul Karger, *IBM T.J. Watson Research Center*
Terence Kelly, *University of Michigan*
Patrick McDaniel, *University of Michigan*
Niels Provos, *University of Michigan*
Gwen Thomas, *Rice University*
Leendert van Doorn, *IBM T.J. Watson Research Center*
David Wagner, *University of California at Berkeley*

The USENIX Association Staff

9th USENIX Security Symposium

August 14–17, 2000
Denver, Colorado, USA

Wednesday, August 16

OS Security
Session Chair: Dan Wallach, Rice University

MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications 1
Anurag Acharya and Mandar Raje, University of California at Santa Barbara

A Secure Java™ Virtual Machine 19
Leendert van Doorn, IBM T.J. Watson Research Center

Encrypting Virtual Memory 35
Niels Provos, University of Michigan

Déjà Vu—A User Study: Using Images for Authentication 45
Rachna Dhamija and Adrian Perrig, University of California at Berkeley

Democracy
Session Chair: Ian Goldberg, University of California at Berkeley

Publius: A Robust, Tamper-Evident, Censorship-Resistant Web Publishing System 59
Marc Waldman, New York University; and Aviel D. Rubin and Lorrie Faith Cranor, AT&T Labs—Research

Probabilistic Counting of Large Digital Signature Collections 73
Markus G. Kuhn, University of Cambridge, U.K.

Can Pseudonymity Really Guarantee Privacy? 85
Josyula R. Rao and Pankaj Rohatgi, IBM T.J. Watson Research Center

Hardware
Session Chair: Markus Kuhn, University of Cambridge, U.K.

An Open-Source Cryptographic Coprocessor 97
Peter Gutmann, University of Auckland, New Zealand

Secure Coprocessor Integration with Kerberos V5 113
Naomaru Itoi, University of Michigan

Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor 129
John Scott Robin, U.S. Air Force; and Cynthia E. Irvine, Naval Postgraduate School

Thursday, August 17

Intrusion Detection
Session Chair: Wietse Venema, IBM T.J.Watson Research Center

Detecting and Countering System Intrusions Using Software Wrappers 145
Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick, NAI Labs

Detecting Backdoors 157
Yin Zhang, Cornell University; and Vern Paxson, AT&T Center for Internet Research at ICSI

Detecting Stepping Stones 171
Yin Zhang, Cornell University; and Vern Paxson, AT&T Center for Internet Research at ICSI

Automated Response Using System-Call Delays 185
Anil Somayaji, University of New Mexico; and Stephanie Forrest, Santa Fe Institute

Network Protection
Session Chair: Tara Whalen, Communications Research Centre Canada

CenterTrack: An IP Overlay Network for Tracking DoS Floods 199
Robert Stone, UUNET Technologies, Inc.

A Multi-Layer IPsec Protocol 213
Yongguang Zhang and Bikramjit Singh, HRL Laboratories, LLC

Defeating TCP/IP Stack Fingerprinting 229
Matthew Smart, G. Robert Malan, and Farnam Jahanian, University of Michigan

E-mail
Session Chair: Elizabeth Zwicky, Counterpane Internet Security

A Chosen Ciphertext Attack Against Several E-Mail Encryption Protocols 241
Jonathan Katz, Columbia University; and Bruce Schneier, Counterpane Internet Security, Inc.

PGP in Constrained Wireless Devices 247
Michael Brown and Donny Cheung, University of Waterloo, Canada; Darrel Hankerson, Auburn University; Julio Lopez Hernandez, State University of Campinas, Brazil, and University of Valle, Colombia; and Michael Kirkup and Alfred Menezes, University of Waterloo, Canada

Shibboleth: Private Mailing List Manager 263
Matt Curtin, Interhack Corporation

MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications

Anurag Acharya, Mandar Raje
Dept. of Computer Science, University of California, Santa Barbara

Abstract

Designing a suitable confinement mechanism to confine untrusted applications is challenging as such a mechanism needs to satisfy conflicting requirements. The main trade-off is between ease of use and flexibility. In this paper, we present the design, implementation and evaluation of *MAPbox*, a confinement mechanism that retains the ease of use of application-class-specific sandboxes such as the *Java* applet sandbox and the *Janus* document viewer sandbox while providing significantly more flexibility. The key idea is to group application behaviors into classes based on their expected functionality and the resources required to achieve that functionality. Classification of application behavior provides a set of labels (e.g., `compiler`, `reader`, `netclient`) that can be used to concisely communicate the *expected functionality* of programs between the provider and the users. This is similar to *MIME-types* which are widely used to concisely describe the *expected format* of data files. An end-user lists the set of application behaviors she is willing to allow in a file. With each label, she associates a sandbox that limits access to the set of resources needed to achieve the corresponding behavior. When an untrusted application is to be run, this file is consulted. If the label (or the MAP-type) associated with the application is not found in this file, it is not allowed to run. Else, the MAP-type is used to automatically locate and instantiate the appropriate sandbox. We believe that this may be an acceptable level of user interaction since a similar technique (i.e., MIME-types) has been fairly successful for handling documents with different formats. In this paper, we present a set of application behavior classes that we have identified based on a study of a diverse suite of applications that includes CGI scripts, programs downloaded from well-known web repositories and applications from the Solaris 5.6 distribution. We describe the implementation and usage of MAPbox. We evaluate MAPbox from two different perspectives: its effectiveness (how well it is able to confine a suite of untrusted applications) and efficiency (what is the overhead introduced). Finally, we describe our experience with MAPbox and discuss potential limitations of this approach.

1 Introduction

Designing a suitable mechanism to confine untrusted applications is a challenging task as such a mechanism needs to satisfy conflicting requirements. The key trade-off is between ease of use and flexibility. To be easy to use, a confinement mechanism should require little or no user input. As a result, such a mechanism is likely to provide one-size-fits-all functionality – that is, all applications being confined are allowed to access exactly the same set of resources. This limits the class of applications that can be used effectively while being confined. To be more flexible, a confinement mechanism has to either allow access to all resources to all applications (which defeats the purpose of confinement) or it has to somehow select the set of resources each application is allowed to access. To be able to select the set of resources that each application is allowed to access, such a mechanism needs *some* knowledge of the application's resource requirements as well as the user's intent.

Previous research into creating confinement environments (also referred to as *sandboxes*) has taken one of four approaches which make different trade-offs between flexibility and ease of use. Several researchers have proposed some form of per-program access control [4, 5, 7, 11, 12, 14, 21]. This approach is highly flexible but requires users (or administrators) to specify access-control information for every program. It can work well if the number of untrusted applications is small and changes infrequently. Several computing environments, however,

are dynamic and contain a large number of applications (e.g., a web-hosting service which allows its users to run CGI scripts). The second approach uses finite-state machine descriptions of program behavior [13, 15, 17]. This provides even more flexibility as different sequences of the same set of accesses can be distinguished. To be used effectively, however, this approach requires a careful understanding of the behavior of individual applications. Given the size, complexity and the number of applications in modern computing environments, it would be hard to develop such detailed descriptions.

The third approach considers each application provider (author/company/web site) as a principal and uses per-provider access-control lists (ACLs) [9, 10, 20]. This groups applications from the same provider into the same sandbox. This is a promising approach since a user needs to deal with potentially fewer principals than the first two approaches. This makes it easier for the users to create and maintain the corresponding ACLs. However, disparate applications from the same provider may be grouped into the same sandbox. To allow *all* of these applications to run, a user may have to provide an overly coarse sandbox – which may or may not be desirable. Another potential problem is that the number of potential providers is large and growing. Creating and maintaining ACLs for a large number of providers can require substantial administrative effort.

The fourth approach consists of special-purpose sandboxes for specific classes of applications, e.g, document viewers [8], applets [6], global computing [3], CGI scripts [18] and programs that run with root privileges [19]. By limiting the scope of the confinement mechanism, these techniques significantly reduce the administration effort required. While each of these sandboxes are easy to use when they are applicable, they are limited in their applicability. For each application, one needs to *manually* find, deploy and instantiate the appropriate sandbox. In addition to being an administrative burden, using a variety of programs for sandboxing makes it harder to check the sandboxes themselves for security flaws.

In this paper, we present the design, implementation and evaluation of *MAPbox*, a confinement mechanism that retains the ease of use of application-class-specific sandboxes while providing significantly more flexibility. The key idea is to group application behaviors into classes based on the expected functionality and the resources required to achieve that functionality. Examples of behavior classes include filters, compilers, editors, browsers, document viewers, network clients, servers etc. Classification of the behavior of an application provides a label (the name of its behavior class) which can be used by its provider to concisely describe its *expected functionality* to its users. This is similar to MIME-types which are widely used to concisely describe the *expected format* of files. We refer to the label assigned to an application as its *Multi-purpose Application Profile*-type (or *MAP-type*). An end-user specifies the set of application behaviors she is willing to allow as a set of MAP-types listed in a `.mapcap` file. With each MAP-type, she associates a suitable sandbox. When an untrusted application is to be run, this file is consulted. If the MAP-type associated with the application is not present in the `.mapcap` file, the application is not allowed to run. Else, the MAP-type is used to automatically locate and instantiate the appropriate sandbox without requiring user intervention. We believe that this may be an acceptable level of user interaction since a similar technique has been fairly successful for handling documents with different formats. For MIME-types, end-users specify, in a `.mailcap` file, which MIME-types they are willing to view, which application is to be used to view MIME-type and how should this application be invoked.

In effect, MAPbox allows the *provider* of a program to *promise* a particular behavior and allows the *user* of a program to confine it to the resources *she* believes are sufficient for that behavior. For CGI scripts provided by users of a web-hosting service, the MAP-type for the script can be specified by the user when it is submitted for installation. For plug-ins and other applications that are downloaded on demand, the MAP-type can be specified in the HTTP header (just as MIME-types are specified for downloaded documents). For applications downloaded and built locally, the MAP-type can be specified by the provider (e.g., in a README file). Note that the provider of a program only specifies the MAP-type for the program, she *does not* specify the sandbox to be used. The association between MAP-types and sandboxes is completely under the control of the user of the program (being specified in the `.mapcap` file in the user's home directory).

This proposal raises several questions. First, can application behaviors be suitably classified? That is, do application behaviors and the corresponding resource requirements fall into distinct categories? Second, how does MAPbox deal with a group of

applications that exhibit similar behavior but need different resources? For example, `hotjava` and `trn` are both browsers that connect to remote servers. However, they differ in the hosts they connect to, the port they connect to and the directory they use to store the downloaded information. Third, how are the individual sandboxes used by MAPbox to be implemented? There are conflicting constraints – on one hand, all accesses must be checked; on the other hand, the overhead should be acceptable. Finally, how well does this approach work in practice?

In section 2, we describe a study of the behavior and resource requirements of fifty applications. These applications were drawn from different sources: CGI scripts downloaded from a well-known CGI repository; programs downloaded from well-known program repositories; and applications provided as part of the Solaris 5.6 environment. Based on this study, we have defined a set of behavior classes and the corresponding sandboxes. In section 3, we present the design and implementation of MAPbox. Our implementation of MAPbox runs on Solaris 5.6 and confines native binaries. It also provides a sandbox description language that can be used to construct new sandboxes with relative ease. In section 4, we describe how MAPbox can be configured and used. In section 5, we present an evaluation of MAPbox. We evaluated both its effectiveness (how well it is able to confine a suite of untrusted applications) and efficiency (what is the overhead introduced). Our results indicate that the overhead of confinement is small enough (< 5% for CGI scripts, 1-33% for other applications) to be acceptable for many applications and environments. We found that a MAP-type-based approach is quite effective for confining untrusted applications. Of the 100 applications in our evaluation suite, only nine failed to complete their test workloads; of these five failed because they made inherently unsafe requests. We also found that mislabeled applications (i.e., applications that were labeled with a different MAP-type than their own) were not able to gain access to resources that the user did not wish to grant. We conclude with a discussion of our experience with MAPbox and the potential limitations of this approach.

2 Identifying Behavior Classes

To identify application behavior classes, we studied a suite of fifty applications. Of these, twenty were Perl-based CGI scripts that we downloaded from a well-known repository; another fifteen were programs downloaded from various well-known repositories; and the final fifteen were applications provided as part of the Solaris 5.6 distribution.[1] We ran each application on a Solaris 5.6 platform with several workloads. For each execution, we obtained a trace of the system-calls made by the application. To collect the system-call traces, we used the `truss` utility. For each system-call, it prints the name, arguments and the return value. As far as possible, we summarized these traces by identifying groups of system-calls and relating them to higher-level operations such as: accessing files, linking libraries, making/accepting network connections, creating child processes, accessing the display, handling signals etc. Figure 1 presents one such group. For other examples, please see [16]. In some cases, to verify the mapping between a higher-level operation and the system-calls it generates, we wrote short programs performing the operation and compared their traces with that of the application being studied.

To design the workloads for our study, we considered two alternative techniques. The first technique starts with an intuitive notion of application behavior classes such as editors, document viewers, compilers, mailers, etc. For each class, it defines a synthetic workload that exercises the *primary* behavior of the class. The second technique develops trace-based workloads by having a set of users to use individual applications and keeping track of user operations for relatively long sessions. Trace-based workloads have the advantage of being more realistic. However, many applications can exhibit multiple behaviors (e.g., `gnu-emacs` can be used as an editor, a news-reader, a mailer etc). Since our goal in this study was to identify the set of resources needed for the individual behaviors, we chose to use synthetic workloads instead of trace-based workloads. For example, for editors, we used the following workload: (1) start up with no file and exit; (2) start up with an existing file and exit; (3) start up with an existing file, delete 100 characters, add 100 characters and exit; (4) for text editors, edit a file, spell-

[1] The CGI scripts were `ads`, `AtDot-2.0.1`, `authentication`, `banner`, `bbs`, `bookofguests`, `bp`, `browsermatcher`, `bsmidi`, `calendar`, `chat`, `counter`, `CrosswordMaker`, `DB_Manager`, `DB_Search`, `dcguest2`, `formmail`, `form_processor`, `guestbook`, and `juke`. All of these are linked off `cgi.resource-index.com`, a well-known CGI repository. The downloaded programs were `idraw`, `xfig`, `ghostview`, `xv`, `gcc`, `pico`, `pine`, `elm`, `lynx`, `agrep`, `xcalc`, `ical`, `xdvi`, `gzip`, `httpd`. The Solaris applications were `vi`, `pageview`, `imagetool`, `dvips`, `mailtool`, `trn`, `Netscape`, `hotjava`, `sh`, `ftp`, `finger`, `rwho`, `whois`, `telnet` and `sed`.

check it and exit; (5) for graphical editors, generate a postscript file and exit. Workloads used for most of the other classes are described in [16].

Based on the results of this study, we identified a set of behavior classes and their resource requirements. We first determined the resources needed by each application. By resources, we mean files, directories, network connections (hosts and ports), the X server, other devices, ability to create new processes, environment variables etc. For each behavior class, we identified resources commonly required to implement the *primary* functionality of the applications in the class. Some applications make use of resources that are not really needed for implementing their primary functionality. For example the Solaris C compiler opens a socket to a license server to check licensing information. Other C compilers (e.g., `gcc`) don't need to make network connections. Based on the Principle of Least Privilege, we do not consider such resources as *requirements* for the corresponding behavior classes. Note that the resource requirements for a class are not simply the union of the resource used by a set of applications that we studied. Instead, they are the set of resources that we believe are *required* to implement the expected functionality for the class. In Section 5, we compare these *expected resource requirements* associated with a behavior class with the *actual resource requirements* of a large suite of applications that implement that behavior.

In addition, we identified a set of *parameters* for each class. Parameters of a class capture common patterns in the idiosyncratic resource requirements of the applications belonging to the class. For example, `hotjava` and `trn` are both browsers that connect to remote servers, download files and present them to users. For this, they need to link in networking libraries, make network connections, open networking-related device files (e.g., `/dev/{tcp,udp,ticotsord}`) and write files in a local directory. However, they differ in the hosts they connect to, the port they connect to and the directory they use to store the downloaded information. In this case, the hosts to connect to, the port to connect to and the directory to store the information would be parameters of the behavior class containing `hotjava` and `trn`.

Table 1 presents the behavior classes we identified and their parameters. We do not claim that the classification presented in Table 1 is either unique or complete. Our goal in identifying these classes was to demonstrate that application behaviors and the corresponding resource requirements *can* be grouped into distinct categories. We expect this classification to be refined based on further experience. This would be similar to the evolution of MIME-types which have been repeatedly refined as users have better understood their potential.

The classes described in Table 1 form a lattice based on their resource requirements. A class `X` is higher in the lattice than a class `Y` if the resources required by `Y` are a proper subset of the resources required by `X`. For example, applications in the `filter` class can access only `stdin/stdout/stderr` whereas applications in the `transformer` class can access `stdin/stdout/stderr` as well as `infile` and `outfile`. We present this lattice in Figure 2.

3 Design and implementation of MAPbox

Our implementation of MAPbox runs on Solaris 5.6 and confines native binaries. We first describe the sandbox description language provided by MAPbox which can be used to construct new sandboxes with relative ease. Next, we describe how MAPbox implements individual sandboxes.

3.1 The sandbox description language

We base our sandbox description language on the configuration language used by Janus [8], a class-specific sandbox for document viewers. Our language consists of eight commands: `path`, `connect`, `putenv`, `rename`, `accept`, `childbox`, `define` and `params`. Figure 3 provides a brief description for these commands (Figure 7 contains a BNF description). Of these, the first four commands were provided by Janus. For a detailed description of these commands, please see [8]. The last four commands are new to MAPbox and are described below. A sample sandbox specification is presented in Figure 9.

accept: this command is the server-side analogue of the Janus `connect` command. It can be used to control the set of peer hosts as well as the set of ports that the confined application can listen on.

```
open("/usr/lib/libsocket.so.1", O_RDONLY)          = 3
fstat(3, 0xEFFFEA00)                               = 0
mmap(0x000000, 8192, PROT_READ, MAP_SHARED, 3, 0)  = 0xEF7B000
mmap(0x000000, 8192, PROT_EXEC, MAP_PRIVATE, 3, 0) = 0xEF7900
close (3)                                          = 0
```

Figure 1: The system-call sequence for dynamically linking a library in Solaris 5.6.

Behavior class	Parameters	Description
`filter`	None	cannot open files, access network/display or exec processes
`reader`	`dir/filelist`	can read files listed in `filelist` or contained in `dir` and its descendants; cannot write files, access network/display or exec processes (e.g, `cat`, CGI scripts that authenticate a user or provide a random image)
`transformer`	`infile, outfile`	can read `infile`, write `outfile`; cannot access network/display or exec processes (e.g., `compress`, `gzip`, image format converters)
`maintainer`	`homedir`	can read and write files in `homedir` and descendants; cannot access network/display or exec processes (e.g., CGI scripts that implement counters, guestbooks, bulletin boards, chat servers, etc.)
`compiler`	`homedir, filelist, libpath, outfile`	can read/write files in `homedir` and descendants; can read files in `filelist`; can read files in all directories on `libpath`; can write `outfile`; cannot access network/display; can exec other applications in the same class (e.g., `gcc`, `tar`, `dvips`, `latex`, `nroff`, `bibtex`, `ld`)
`editor`	`homedir, filelist`	can read/write files in `homedir` and its descendants; can read/write files in `filelist`; cannot access network; can access display; can exec applications labeled `filters` or `transformers` (e.g., `gnu-emacs`, `vi`, `pico`, `xfig`, `idraw`)
`viewer`	`homedir, filelist`	can read/write files in `homedir` and its descendants; can read files in `filelist`; cannot access network; can access display; can exec applications in the same class (e.g., `ghostview`, `pageview`, `imagetool`, `xdvi`)
`netclient`	`host, port, dir`	can connect to `host` at `port`; can read and write files in `dir`; cannot exec processes; cannot access display (e.g., `ftp`, `finger`, `wget`)
`mailer`	`homedir, [mailbox], [gateway], [mailcommand]`	can read/write files in `homedir` and descendants, can read/write `mailbox` file (if specified), can connect to `gateway` (if specified) on port 25; can access display; can exec `viewers` and `filters`, can exec the `mailcommand` (if specified) (e.g., `pine`, `elm`, `mailtool`, many CGI scripts that implement guestbook, mailing lists and bulletin boards)
`browser`	`homedir, filelist, hostlist, port`	can read/write files in `homedir` and descendants; can read files in `filelist`; can connect to hosts in `hostlist` at `port`, can access display; can exec `viewers` (e.g., `lynx`, `hotjava`, `trn`)
`netserver`	`homedir, hostlist, port`	can read/write files in `homedir` and descendants; can accept connections at `port` from hosts in `hostlist`; cannot access display; can exec `filters`, `transformers` and `maintainers` (e.g., `httpd`, `ftpd`)
`shell`	`path, mapfile, [maptypelist]`	can exec binaries found in the directories listed in `path`; can read `mapfile`; `maptypelist` can be used to limit the MAP-types of applications that be exec'ed; cannot access network; cannot access display (e.g., `ksh`, `csh`, `tcsh`)
`game`	`homedir`	can read/write `homedir`; can access display; cannot access network; can exec applications in the same class
`applet`	`host, port, path`	can access display; can connect to `host` at `port`; can read files in directories listed in `path`; cannot write files; cannot exec processes.

Table 1: Brief descriptions of the behavior classes identified in this study.

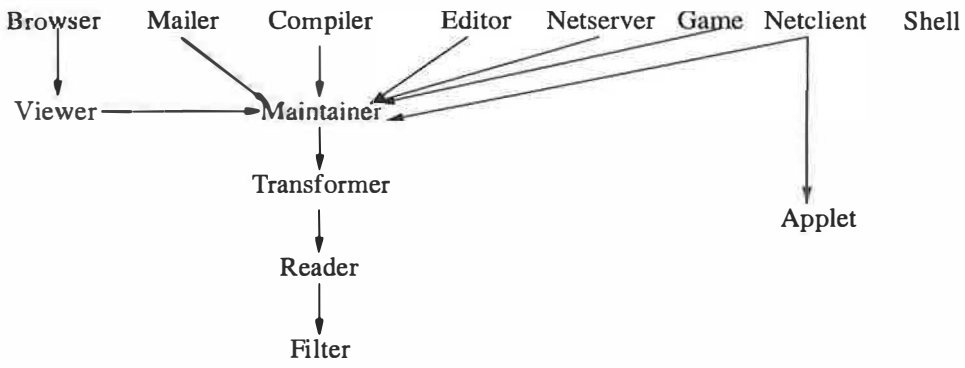


Figure 2: Lattice describing the relationship between application behavior classes.

Command	Description
`path`	used to `allow` or `deny` `read/write/exec` access to a list of files (e.g., `path deny read,write,exec /etc`). Wildcards are allowed; relative paths are not allowed; `deny` takes precedence over `allow`.
`rename`	used to redirect accesses to a particular file to a different file (e.g., `rename read /etc/passwd /tmp/dummy`). Wildcards and relative paths are not allowed.
`connect`	used to control connections to remote hosts and the X server. Must be specified as IP addresses; wildcards allowed (e.g., `connect allow tcp 128.111.*.*:80/128.32.*.*:8080`)
`putenv`	used to add a variable definition to the environment (e.g., `putenv HOME=/tmp/boxedin`)
`accept`	used to control connections from remote hosts (e.g., `accept allow udp 128.111.*.*:513`)
`childbox`	used to specify the sandbox to be used for processes forked by the confined application (e.g., `childbox viewer`). At most one `childbox` command allowed per sandbox.
`define`	used to define a symbolic value that can be used later (e.g., `define _NETWORK_FILES /etc/netconfig /etc/nsswitch.conf /etc/.name_service_door`)
`params`	used to define the parameters for a sandbox (e.g., `params infile outfile`). Parameters are refered to using a $ prefix (e.g., `$outfile`)

Figure 3: Brief description of the MAPbox sandbox description language.

The value `NON_SYSTEM_PORT` can be used to indicate any port not reserved for system services.

childbox: this command is used to specify a different sandbox for the processes forked by the confined application. If no childbox command occurs in a sandbox specification, the original sandbox is used to confine the children, if any. For example, children of `browsers` can be restricted to be `viewers`.

define: this command can be used to define symbolic constants which can then be used in other commands. Symbolic constants can be used to simplify the task of porting sandboxes across platforms. For example, to be able to access the network on many platforms, an application needs to link in a platform-dependent set of libraries[2] and read a platform-dependent set of configuration files.[3] Symbolic definitions can be used to isolate these dependencies. As long as the sandboxes are defined in terms of symbolic constants which are collected in a single file, porting the entire set of sandboxes is a matter of redefining the symbolic constants in this file. To support this, MAPbox reads a common specification file before it reads the specification file for a particular sandbox. Figure 8 presents an example of a common specification file for Solaris 5.6.

params: this command is used to define the parameters for a sandbox. This command can occur only once in a specification and must precede all other commands.

[2] On Solaris 5.6, `/usr/lib/libsocket.so.1`, `/usr/lib/libnsl.so.1.so.1` and `/usr/lib/nss_compat.so.1`.

[3] On Solaris 5.6, `/etc/netconfig`, `/etc/nsswitch.conf` and `/etc/.name_service_door`.

3.2 Implementation details

Initialization: MAPbox starts by reading the sandbox specification file (specified on the command line) and building the `Policy` structure. The `Policy` structure consists of eight components: (1) `read-list` (list of files that can be read), (2) `write-list` (list of files that can be written), (3) `exec-list` (list of binaries that can be `exec`'ed), (4) `rename-list` (list of files whose accesses are to be redirected to some other file), (5) `connect-list` (list of host/port combinations that the confined application can connect to), (6) `accept-list` (list of hosts that the confined application can accept connections from and the ports that it can bind to), (7) `env-list` (list of environment variables for the confined application), and (8) `childbox` (the sandbox to be used for child processes, if any). It first forks. The forked version sets up the environment for the application to be confined by: limiting the environment variables to those specified in sandbox, setting umask to 077, limiting the virtual memory use and datasize, disabling core dumps, changing the current working directory to the application's `homedir` directory,[4] and closing all unnecessary file descriptors. It then `exec`'s the application to be confined.

Interception mechanism: we use the `/proc` interface provided by Solaris 5.6 to intercept selected system-calls. The `/proc` interface has been previously used by researchers for building class-specific sandboxes [3, 8] and for user-level extensions to operating systems [2]. This interface guarantees that *all* system-calls are intercepted. It allows us to intercept system-calls both on their entry to and exit from the operating-system. The interception mechanism provides information about the identity of an intercepted system-call, its arguments, whether it is an entry or an exit, and the return value (if intercepted on exit). We intercept most system-calls on their entry to the kernel to allow or deny access to resources; we intercept a few system-calls on their return from the kernel to record a returned value (e.g., `fork`) or to control access to blocking communication calls (e.g., `accept` for which the identity of the peer is known only when it returns). MAPbox maintains a handler for every intercepted system-call (separate handlers are maintained for entry and exit). When a system-call is intercepted, the corresponding handler is invoked. To deny a system-call, the handler sets a field in the structure used to communicate between the kernel and MAPbox. A denied system-call returns to the application with an error code of `EINTR`. For a description of individual system-call handlers, please see [16].

Handling symbolic links: since Unix file-systems support symbolic links, simply checking the arguments for file-system-related system-calls is not sufficient to implement file-system-related checks. For example, `/tmp/letter-to-my-mom.txt` can be a symlink to `/etc/passwd`. To plug this hole, MAPbox completely resolves each filename (using the `resolvepath()` call available in Solaris) before checking it against the `Policy` structure.

Redirecting requests for sensitive files: to redirect requests for a sensitive file to a benign dummy file, MAPbox resolves all filenames completely and compares them with the completely resolved name of the sensitive file. If a match is found, it writes the name of the dummy file as a string on the stack of the confined process,[5] and changes the pointer to the filename argument to the intercepted system-call to point to this string. It then allows the system-call to proceed.

Confining child processes: MAPbox creates a separate copy of itself for every child of a confined process. To achieve this, it intercepts the `fork` system-call on exit and extracts the process-id of the newly created process. It then forks itself and attaches the child MAPbox process to the newly created application process. Unless specified otherwise, the child of a confined application is confined in the same sandbox as the parent. If, however, a different sandbox is specified (using the `childbox` command), the instance of MAPbox corresponding to the child process intercepts the subsequent `exec` system-call and reads the appropriate sandbox specification file.

Other system-calls: the MAPbox sandbox specification language can specify the confinement requirements for most but not all system-calls. For the remaining system-calls, MAPbox implements an

[4] If no `homedir` directory exists for the application, a temporary directory is created in `/tmp` for this execution and is used as the current working directory. This directory is deleted after the confined program terminates.

[5] On Solaris 5.6, this is implemented using `pwrite()` and `ioctl()`s on the `/proc` file corresponding to the confined application.

application-independent policy. It does not intercept system-calls related to signals, threads and virtual memory. For these resources, it relies on the security provided by the kernel. It also does not intercept system-calls that perform read/write or send/receive operations – depending on the checks performed for initializing operations such as `open()`, `creat()`, `socket()` etc. For others, it takes a conservative approach and denies all system-calls that it does not know to be safe.

- it denies calls that can be invoked only with super-user privileges (e.g., `mount`, `umount`, `plock`, `acct`, etc.).
- it currently denies calls to `acl()` which gets/sets the access-control list for a file. We have not yet seen these system-calls in traces.
- it denies all calls to `door()` except those used to query the host database.
- it allows `fcntl()` calls with F_DUPFD, F_DUP2FD (which return new file descriptors) and F_GETFD, F_SETFD (which read and write file descriptor flags) commands. It denies `fcntl()` calls with other commands.
- it allows a small number of `ioctl` calls on `stdin` and `stdout`. It currently denies all other `ioctl` calls. This call performs a variety of control functions on devices and streams. Properly handling `ioctl` requires a good understanding of the individual devices and their controls.

Confining X applications: The X protocol has been designed for use by cooperative clients. Any client application is able to manipulate or modify objects created by any other client application run by the same user.[6] This has been done for two reasons. First, it allows window managers to be written as ordinary clients and second, it allows clients to communicate to implement cooperative functionality such as cut-and-paste.

To confine X applications, we have developed `Xbox`, an X protocol filter [1]. `Xbox` has been designed to be used in conjunction of a system-call-level sandbox such as MAPbox and Janus and is to be interposed between an untrusted application and the X server. Before starting an untrusted X application, MAPbox sets the DISPLAY environment variable to a socket that `Xbox` listens on (`unix:4` by default). It then makes sure that the confined application does not bypass `Xbox` by denying direct connections to the X server.

`Xbox` snoops on all protocol messages and keeps track of the resources (windows, pixmaps, cursors, fonts, graphic contexts and colormaps) created by the confined application. `Xbox` can be easily extended to handle extensions to the X protocol. The current implementation handles the `SHAPE`, `MIT-SCREEN-SAVER`, `DOUBLE-BUFFER`, `Multi-Buffering`, and `XTEST` extensions. The confined application is allowed to access/manipulate only the resources that it has created and is allowed to read limited information from the root window (the operations it allows on the root window are both necessary and safe). All other requests regarding specific resources are denied (e.g., CreateWindow, ChangeWindowAttributes, GetWindowAttributes, InstallColorMap,ReparentWindow, ChangeGC, ClearArea, PolyPoint etc). In addition, the confined application is not allowed to query parts of the window hierarchy it did not create and is allowed limited versions of some operations that change the global state of the server (GrabKey, GrabButton etc). Other global operations (such GrabServer, SetScreenSaver, ChangeKeyboardMapping etc) are denied. Finally, the confined application is not allowed to communicate with other applications via the X server.

4 Configuration and administration

There are two ways in which MAPbox can be configured. First, by listing the MAP-types allowed by the user in a `.mapcap` file; and second, by placing commands in a site-wide specification file which MAPbox reads when it starts up.

Specifying acceptable MAP-types: the list of MAP-types acceptable to the user can be specified in a `.mapcap` file. This file contains a sequence of entries consisting of (MAP-type, sandbox-file) pairs. A MAP-type consists of the name of a behavior class with values for all its parameters. The corresponding sandbox file contains a description of the sand-

[6] The existing security mechanisms provided by the X server, i.e., the `xhost`-based mechanism and the `xauthority`-based mechanism cannot distinguish between multiple applications belonging to the same user.

box that is to be used for this MAP-type. A parameter can be specified using as a symbolic value, a concrete value, a regular expression, a numeric range, or a list. Multiple combinations of parameter values can be specified using separate entries. Parameters for some behavior classes (e.g., `transformer`) include command-line arguments that will supplied only when an application runs (for `transformer`, the the input and output files). These parameters are specified by the meta-values `%a1`, `%a2`, `%a3` etc. These correspond to the arguments supplied to the program – in the same order as they are specified. Several behavior classes have a `homedir` parameter which specifies the home directory for the application. Typically, this is the directory in which all the files for the application reside and the application is allowed to read/write files in this directory and its descendants. To refer to the directory that the binary for an application lives in, MAPbox provides the meta-value `%h` (`h` for `homedir`).[7] The syntax for `.mapcap` entries is presented in Figure 4. A sample `.mapcap` file is presented in Figure 5.

To check if an application is to be allowed to run, the MAP-type specified by the provider is matched against entries in the `.mapcap` file. The rules for matching are:

- an empty argument can only be matched by an empty argument.
- meta-values, like `%a1`, `%a2` and `%h`, can be matched only by themselves.
- for all other arguments, the value provided by the application provider should not be more general than the value in the `.mapcap` file. For example, `browser(%h,www.aol.com,80)` would match the specification in the `.mapcap` file in Figure 5 whereas `browser(%h,*,*)` would not.

Implementing site-wide policies: as mentioned in Section 3, MAPbox reads a common specification file before it reads the specification file for a particular sandbox. In addition to making sandbox specification files more portable, this feature can also be used to implement site-wide policies. The purpose of this feature is not to deal with malicious users – it is easy to bypass this mechanism. Instead it is to rapidly respond to problems in a cooperative environment. Figure 8 contains a sample of a common specification file.

[7] Executables in a a software package are often placed in a "appDir/bin" directory whereas the resource files are usually placed in a separate subdirectory of "appDir" (e.g. "appDir/lib"). To handle this common case, MAPbox checks if the last element in an application's pathname is "bin". If so, it removes this element. For example, if the application lives in "/apphome/bin", this meta-value would expand to "/apphome".

5 Evaluation of MAPbox

We evaluated MAPbox from two different perspectives: its effectiveness (how well it is able to confine a suite of untrusted applications) and efficiency (what is the overhead introduced).

5.1 Effectiveness of MAPbox

For these experiments, we used a suite of 100 applications: the fifty applications used in the application characterization study mentioned in Section 2 and fifty additional applications. Of the additional applications, twenty were Perl-based CGI scripts from *cgi.resource-index.com*, fifteen were programs that we downloaded from different repositories and built locally and fifteen were applications from the Solaris 5.6 distribution.[8] We assigned each application a MAP-type based on the code (where available), the associated documentation (manual, man page, README file) and a trace of the system calls it makes.

We performed two sets of experiments. The first set of experiments were designed to check if the behavior classes identified in Section 2 were too restrictive. In other words, is MAPbox so restrictive that few or no applications can be successfully run while confined? The second set of experiments were designed to check if the behavior classes were too broad. That is, is MAPbox so lax that mislabeled applications (i.e., applications that were labeled with a different MAP-type than their own) are able to gain access to

[8] The CGI scripts were `jchat10c`, `kewlcheckers`, `kewlchess`, `mazechat`, `multimail`, `netcard201`, `picpost`, `postit`, `robpoll`, `SDPGuestbook`, `SDPMail`, `SDPUpload`, `search`, `showsell`, `UltraBoard_1.62`, `web_store`, `webadverts`, `webbbs`, `webodex`, `wwwchat30`. The downloaded and built programs were `gnu-emacs`, `lcc`, `javac`, `wget`, `ksh`, `latex`, `bibtex`, `xbiff`, `xclock`, `groff`, `gnuplot`, `mpeg_play`, `cjpeg`, `gzcat`, `md5sum`. The Solaris 5.6 applications were `tcsh`, `comm`, `detex`, `deroff`, `compress`, `tar`, `ld`, `talk`, `strings`, `sort`, `diff`, `s2p`, `find2perl`, `mpage` and `cc`.

```
entry          := behaviorClass (args) sandboxfile
behaviorClass  := filter | transformer | ...
args           := /* empty */ | arg | args arg , arg
arg            := value | list | %a | %c | /* empty */
list           := {values}
values         := values , value | value
value          := regexp | [ num - num ]
```

Figure 4: Syntax for `.mapcap` entries.

```
filter()                 /fs/play/~user/mapbox/sandboxes/filter.box
transformer(%a1,%a2)     /fs/play/~user/mapbox/sandboxes/transformer.box
browser(%h,*,80)         /fs/play/~user/mapbox/sandboxes/browser.box
game(%h)                 /fs/play/~user/mapbox/sandboxes/game.box
maintainer(%h)           /fs/play/~user/mapbox/sandboxes/maintainer.box
```

Figure 5: Sample `.mapcap` file.

resources that the user did not wish to grant? For the first set of experiments, we ran each application within the sandbox associated with its own MAP-type. For the second set of experiments, we ran each application within a sandbox that corresponds to a MAP-type other than its own. For both experiment sets, we ran these applications with workloads similar to those used in the classification study described in Section 2.

5.1.1 Is MAPbox too restrictive?

Of the 100 applications in our evaluation suite, only nine failed to complete their workload when run within the sandbox for their own MAP-type. Of these, six belonged to the original set of 50 applications that were used in the classification study described in Section 2, the remaining three belonged to the second set of 50 applications added for these experiments.[9] Of the 40 CGI scripts in the suite, one failed; of the 30 downloaded programs, five failed; of the 30 Solaris applications three failed. Of these nine programs, five (`xv`, `xfig`, `pageview`, `lynx` and `Netscape`) failed because they made unsafe accesses and the other four failed inspite of making accesses that we manually verified to be safe. Of the latter, two (`gcc` and `gnu-emacs`) failed because they made a sequence of requests that were individually unsafe but taken as a sequence, implement a safe operation. Since MAPbox makes decisions about each system-call independently, it is unable to detect such cases. The last two, (`cc` and `multimail`) failed because they do not fit into our current collection of MAP-types.

Applications that failed due to unsafe operations:: Three applications failed because they tried to perform unsafe X window operations: `xv` failed when it tries to scan the entire window hierarchy of the X server; `xfig` failed trying to allocate a colormap not owned by itself; and `pageview` failed trying to change an attribute of a window not owned by itself. Two other applications failed because they were denied access to sensitive files: `lynx` tried to access the password database via a `door()` call; `Netscape` needed access to non-empty `/etc/passwd` and `/etc/mnttab`.

Applications that failed due to local nature of checking: Several applications try to determine the current working directory, a safe operation by itself, by walking up the directory hierarchy using relative paths, which is an unsafe operation. Figure 6 illustrates this behavior using a system-call trace excerpt. MAPbox does not allow this operation since it denies all file-system calls with relative paths. Two applications, `gcc` and `gnu-emacs`, failed due to this limitation. Another application, the Solaris C compiler `cc`, also failed while performing this operation but had another reason for failure (see below). Note that this particular problem can be elim-

[9] As mentioned in Section 2, the resource requirements for a class are not simply the union of the resource used by a set of applications that we studied. Instead, they are the set of resources that we believe are *required* to implement the expected functionality for the class.

inated if the Solaris system-call interface is extended to provide a *getcwd()* operation directly. However, the general problem of not being able to distinguish safe sequences of potentially unsafe operations is inherent to the MAP-box approach. Based on our experience, however, we expect this problem to be rare.[10]

Applications that failed due to lack of a suitable MAP-type: Two applications failed as they could not fit into our current collection of MAP-types: `cc` (the Solaris C compiler), and `multimail` (a CGI mailing program). The Solaris C compiler fails because it connects to a license server and the sandbox for a `compiler` does not allow access to the network. If desired, this can be fixed by introducing a new MAP-type, say `licensed-compiler`, which includes the host and port number of the license server as parameters. The CGI mailer, `multimail`, fails as it invokes a program (`/bin/date`) that is not the mail command. If desired, this problem can be fixed by rewriting the program to directly determine the current time.

Note that only four applications from a diverse suite of 100 applications fail due to features of MAPbox. This indicates that a MAP-type-based approach is not too restrictive.

5.1.2 Is MAPbox too lax?

For each application used in these experiments, we selected a conflicting MAP-type, that is, a MAP-type that would allow the application to access resources that it would not be allowed to if correctly labeled. In effect, we picked a MAP-type that was not its own and was not an ancestor in the lattice shown in Figure 2. Of the 100 applications in our evaluation suite, not one completed its workload in these experiments. This provides evidence that MAPbox is not too lax.

5.2 Efficiency of MAPbox

To evaluate the efficiency of the MAPbox implementation, we ran two sets of experiments. In the first set, we used MAPbox to confine CGI scripts in a web-server environment and measured the additional latency experienced by web clients over a long-haul network. For these experiments, we used a suite of six CGI scripts. In the second set of experiments, we used MAPbox to confine non-interactive applications in a desktop environment and measured the increase in their execution time. For these experiments, we used a suite of six applications.

The applications used in these experiments and the corresponding workloads are listed in Table 2. We ran each application with and without MAPbox and measured the difference in end-to-end execution time. For each experiment, we also kept track of the time spent in MAPbox code. We conducted these experiments on a lightly loaded SUN Ultra-1/170 with 64 MB and Solaris 2.6 (i.e., the applications and the CGI scripts were run on this machine). All files involved in these experiments were in the OS file-cache. We used the Solaris high resolution timer `gethrtime()` for all measurements.

For the experiments involving CGI scripts, the server (Apache 1.0.2) was at the University of California, Santa Barbara on the US west coast and the client was at the University of Maryland, College Park on the US east coast. We ran these experiments between 1am and 3am Pacific Time when network congestion is usually light. Measurement of the end-to-end execution time was done at the client. The round-trip time between these sites (as determined by `ping`) was about 80 ms. To factor out the effects of transient congestion, we repeated each experiment 100 times and reported the minimum value as the result. For the experiments involving local applications, we repeated each experiment five times and reported the minimum value as the result.

Table 3 presents the results of all experiments. The overhead caused by MAPbox for CGI scripts was small ($< 5\%$) in all experiments. This is to be expected since only a small fraction of the end-to-end execution time in these cases was due to the execution of the scripts themselves; network latency, transfer time and other administrative costs (web server overhead, CGI invocation etc) contributed a large fraction of the execution time. The overhead caused by MAPbox for local applications varied greatly – from about 1% for `gzip-1MB` and `grep` to 33% for `gzip-8KB`. For five out of the six applications, the overhead was below 20%. From these results, we conclude that the overhead of confinement is likely to be acceptable for many applications and

[10] In case, we are mistaken in this expectation, it is quite easy to extend MAPbox to handle relative paths by using the `resolvepath()` system-call to completely resolve all relative paths.

```
stat64("./", 0xEFFFC620)                    = 0
stat64("/", 0xEFFFC588)                     = 0
open64("./../", O_RDONLY|O_NDELAY)          = 3
fcntl(3, F_SETFD, 0x00000001)               = 0
fstat64(3, 0xEFFFBC30)                      = 0
fstat64(3, 0xEFFFC620)                      = 0
getdents64(3, 0x0005A014, 1048)             = 608
close(3)                                    = 0
open64("./../../", O_RDONLY|O_NDELAY)       = 3
fcntl(3, F_SETFD, 0x00000001)               = 0
fstat64(3, 0xEFFFBC30)                      = 0
fstat64(3, 0xEFFFC620)                      = 0
getdents64(3, 0x0005A014, 1048)             = 280
close(3)                                    = 0
```

Figure 6: System-call trace excerpt illustrating the `getcwd()` pattern.

application	type	workload	application	type	workload
`ftp`	local	ftp 10 32KB files from `localhost`	`dvips`	local	convert a 50 page DVI file to postscript
`latex`	local	compile 5 tex files ($\approx$ 300 lines each)	`grep`	local	search gcc source for "int", 182 files
`gzip-1MB`	local	compress 4 1MB files	`gzip-8KB`	local	compress 32 8KB files
`guestbook`	CGI	post 100 1KB msgs	`wwwchat30`	CGI	post 100 128 byte msgs
`counter`	CGI	100 counter accesses	`kewlcheckers`	CGI	make 20 moves
`SDPUpload`	CGI	upload 10 64KB files	`webbbs`	CGI	post 32 8KB msgs

Table 2: Workloads used in the experiments. The two gzip workloads were selected to compare the overheads for processing a few large files with the overhead for processing many small files.

environments.

To determine the cause of the variation in the overhead for local applications, we analyzed their operation in greater detail. We found that the cost of using MAPbox depended on the frequency of file-system-related system-calls (`open`/`stat` etc). To obtain a fine-grain breakdown of this overhead, we added probes in the handlers for these calls and repeated the experiments. We found that most of this overhead (90% of the time spent in MAPbox) is due to two operations: (1) the *resolvepath* operation which is used to safely handle symbolic links by completely resolving a filename (65% of the time spent in MAPbox); and (2) reading the string containing the filename from the confined process's memory (25% of the time spent in MAPbox). These costs are inherent to the system-call interception technique and cannot be eliminated.

6 Discussion

We first present our experience with determining suitable MAP-types for applications. We then discuss potential limitations of the MAPbox approach.

6.1 Experience determining MAP-types

Of the 100 applications in our suite, 91 applications completed their test workloads. Of these, twenty applications were labeled `mailer`, nineteen were labeled `maintainer`, nine were labeled `compiler`, eight each were labeled `reader` and `transformer`, seven each were labeled `netclient` and `viewer`, six were labeled `editor`, three were labeled `shell`, two were labeled `browser` and one each were labeled `filter` and `netserver`.

application	total time	total time with MAPbox	time in MAPbox	other overhead
`ftp`	1.99s	2.32s (17%)	0.17s (9%)	0.16s (8%)
`dvips`	2.88s	3.26s (13%)	0.11s (4%)	0.27s (9%)
`latex`	2.80s	3.06s (9%)	0.17s (6%)	0.09s (3%)
`grep`	2.72s	2.76s (1.2%)	0.02s (0.6%)	0.02s (0.6%)
`gzip-1MB`	4.26s	4.30s (1%)	0.01s (0.2%)	0.03s (0.8%)
`gzip-8KB`	1.52s	2.02s (33%)	0.23s (15%)	0.27s (18%)
`guestbook`	49.1s	51.2s (2.2%)	0.36s (0.7%)	0.74s (1.5%)
`wwwchat30`	19.21s	19.62s (2%)	0.2s (1%)	0.22s (1%)
`counter`	25.4s	26.0s (2%)	0.32s (1%)	0.28s (1%)
`kewlcheckers`	16.94s	17.23s (1.7%)	0.1s (0.6%)	0.19s (1.1%)
`SDPUpload`	22.31s	22.9s (2.6%)	0.3s (1.3%)	0.29s (1.3%)
`webbbs`	26.12s	26.94s (3%)	0.31s (1%)	0.51s (2%)

Table 3: MAPbox overheads. All percentages are with respect to end-to-end execution time without MAPbox (second column). The time in the "other overhead" column includes kernel overhead for intercepting system-calls as well as the cost of the context-switches required to pass information between the kernel and MAPbox.

All the CGI scripts that we studied fell into only four MAP-types: `reader`, `maintainer`, `mailer` and `compiler`.[11] While we expected the first two MAP-types to be common among CGI scripts, the number of scripts that are able to send email was a surprise to us. Seventeen of the forty CGI scripts used in this study invoke `sendmail` and/or open a socket to a mail gateway. This included guestbooks, advertisement managers, homepage providers, web-based rolodex and bullletin board programs. These programs used email to notify users/administrators about events of interest. It appears that the authors of CGI scripts prefer to send mail for this purpose instead of writing to a log file (as is common in conventional applications).

6.2 Potential limitations

We believe that the MAPbox approach provides a good tradeoff between ease of use and flexibility. Nevertheless, it has several potential limitations.

Applications limited to single behavior: the MAPbox approach limits each application to a single behavior. Many applications, however, exhibit multiple behaviors (e.g., Netscape can be used as a browser, mailer and newsgroup reader). In some specific cases, it may be possible to create customized classes that allow a particular group of behaviors. In general, however, we believe this is an inherent trade-off between security and functionality: many applications cannot be securely confined in all their generality.

Confining `setuid root` programs: the system-call interception mechanism used by MAPbox does not work for `setuid root` programs. This is a necessary restriction as allowing a user-level process to intercept the system-calls of a `setuid root` program would provide a trivially easy way to become `root`.

Lack of standardized behavior classes: given that individual end-users are allowed (though not required) to create new MAP-types and the corresponding sandboxes, it is conceivable that everyone defines different MAP-types or uses different names for the same classes resulting in configuration chaos. While this is a possibility, we believe that it is unlikely to happen. As evidence, we point to the web community's experience with MIME-types which have a similar potential for configuration chaos. Instead, the set of MIME-types used by most users has converged to a more-or-less stable set.

Portability: MAPbox depends on the ability to intercept all system-calls for implementing a se-

[11] Only one application was assigned the `compiler` MAP-type: `search` which compiles an index for all files on a website and looks it up on demand.

cure reference monitor. Currently, only Solaris and Linux provide this facility.

7 Conclusions

In this paper, we have presented the design, implementation and evaluation of MAPbox, a confinement mechanism that retains the ease of use of application-class-specific sandboxes, such as Janus, while providing significantly more flexibility. Based on a study of a diverse set of applications, we have identified a set of behavior classes which have intuitive meaning and whose resource requirements can be differentiated. We do not claim that this classification is either unique or complete. Our goal in identifying these classes was to demonstrate that application behaviors and the corresponding resource requirements *can* be grouped into distinct categories. We expect this classification to be refined based on further experience.

To evaluate the effectiveness of MAPbox, we tried to confine a large suite of applications (including Perl-based CGI scripts, downloaded programs and applications from the Solaris distribution) using suitable class-specific sandboxes. We found that a MAP-type-based approach is quite effective for confining untrusted applications. Of the 100 applications in our evaluation suite, only nine failed to complete their test workloads when run within the sandbox corresponding to their own MAP-type. Of the 40 CGI scripts in the suite, one failed; of the 30 downloaded programs, five failed; of the 30 Solaris applications, three failed. Of these, five failed because they made unsafe accesses and only four failed in spite of making accesses that we manually verified to be safe. We also found that mislabeled applications (i.e., applications that were labeled with a different MAP-type than their own) were not able to gain access to resources that the user did not wish to grant.

To evaluate the efficiency of the MAPbox implementation, we ran two sets of experiments – one set with CGI scripts and the other with local applications. We found that the overhead caused by MAPbox for CGI scripts was small (< 5%) in our experiments. The overhead caused by MAPbox for local applications varied greatly – from about 1% to 33%. For five out of the six applications, the overhead was below 20%. We found that the cost of using MAPbox depended on the frequency of file-system-related system-calls. From these results, we conclude that the overhead of confinement is likely to be acceptable for many applications and environments.

Acknowledgments

We would like to thank the authors of Janus for making their implementation available. While MAPbox has been implemented anew and contains much functionality not provided by Janus, we benefited *greatly* from reading their code as well as its lucid description in [8]. We would also like to thank Paul Kmiec for suggesting the use of `pwrite()` to write a string into the stack of a confined process.

References

[1] A. Acharya. The Xbox distribution. Available at *http://www.cs.ucsb.edu/~acha/-software/xbox.tar.gz*, 1999. `Xbox` is a confining filter for X11 applications.

[2] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Extending the operating system at the user level: the Ufo global file system. In *Proc. of the 1997 USENIX Technical Conference.*

[3] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations. Available at *http://www.cs.ucsb.edu/~berto/papers/99-usenix-consh.ps*, Dec 1998.

[4] W. Boebert, R. Kain, W. Young, and S. Hansohn. Secure Ada Target: Issues, System Design, and Verification. In *Proc. of 1985 IEEE Symposium on Security and Privacy*, pages 176–83.

[5] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proc. of the Fifth ACM Conference on Computer and Communications Security*, 1998.

[6] J. Fritzinger and M. Mueller. Java security. Technical report, Sun Microsystems, Inc, 1996.

[7] T. Gamble. Implementing execution controls in Unix. In *Proc. of the 7th System Administration Conference*, pages 237–42, 1993.

[8] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *Proc. of the 1996 USENIX Security Symposium*, 1996.

[9] L. Gong. New security architectural directions for Java. In *Proc. of IEEE COMPCON'97*, 1997.

[10] T. Jaeger, A. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proc. of the Sixth USENIX Security Symposium*, 1996.

[11] P. Karger. Limiting the damage potential of the discretionary trojan horse. In *Proc. of the 1987 IEEE Syposium on Security and Privacy*, 1987.

[12] M. King. Identifying and controlling undesirable program behaviors. In *Proc. of the 14th National Computer Security Conference*, 1992.

[13] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings. 10th Annual Computer Security Applications Conference*, pages 134–44, 1994.

[14] N. Lai and T. Gray. Strengthening discretionary access controls to inhibit trojan horses and computer viruses. In *Proc. of the 1988 USENIX Summer Symposium*, 1988.

[15] N. Mehta and K. Sollins. Extending and expanding the security features of Java. In *Proc. of the 1998 USENIX Security Symposium.*

[16] M. Raje. Behavior-based confinement of untrusted applications. Technical Report TRCS99-12, Dept of Computer Science, University of California, Santa Barbara, Jan 1999.

[17] F. Schneider. Enforceable security policies. Technical report, Dept of Computer Science, Cornell University, 1998.

[18] L. Stein. SBOX: put CGI scripts in a box. In *Proc. of the 1999 USENIX Technical Conference.*

[19] K. Walker, D. Sterne, M. Badger, M. Petkac, D. Shermann, and K. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proc. of Sixth USENIX Security Symposium*, 1996.

[20] D. Wallach, D. Balfanz, D. Dean, and E. Felten. Extensible security architecture for Java. In *Proc. of the Sixteenth ACM Symposium on Operating System Principles*, 1997.

[21] D. Wichers, D. Cook, R. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo. PACL's: an access control list approach to anti-viral security. In *USENIX Workshop Proceedings. UNIX SECURITY II*, pages 71–82, 1990.

```
command       :=   path_c | rename_c | connect_c | accept_c | putenv_c | childbox_c
path_c        :=   path permission access_modes file_list
rename_c      :=   rename file1 file2
connect_c     :=   connect permission protocol ip_addr_list
                |  connect permission display
accept_c      :=   accept permission protocol ip_addr_list
putenv_c      :=   putenv name_val_list | putenv DISPLAY
childbox_c    :=   childbox class
permission    :=   allow | deny
access_modes  :=   access_modes , access_modes | access_mode
access_mode   :=   read | write | exec
file_list     :=   filename file_list | filename
protocol      :=   tcp | udp | *
ip_addr_list  :=   ip_addresses : port_addr
ip_addresses  :=   ip_addr , ip_addresses | ip_addr | *
port_addr     :=   port_num / port_mask | port_num | *
```

Figure 7: Grammar for the sandbox description language. Note that the `define` and `params` commands are not included in the above description. These commands are implemented as macros in a preprocessing step.

```
define _COMMON_LD_LIBRARY_PATH /usr/openwin/lib:/usr/ucblib

define _COMMON_READ /dev/zero /usr/lib/locale/*

# /dev/zero is a device file used for mmap's
define _COMMON_WRITE /dev/zero
# this is true in our environment
define _COMMON_TERM  xterm
# redirect X requests to the Xbox filter
define _COMMON_DISPLAY unix:4

define _COMMON_LIBS /usr/lib/libthread.so.1 /usr/lib/libICE.so.6\\
/usr/lib/libSM.so.6 /usr/lib/libw.so.1 /usr/ucblib/* \\
/usr/lib/libc.so.1 /usr/lib/libdl.so.1 /usr/lib/libintl.so.1\\
/usr/lib/libelf.so.1 /usr/lib/libm.so.1 /usr/lib/liballoc.so.1\\
/usr/lib/libmp.so.2 /usr/lib/libmp.so.1 /usr/lib/libsec.so.1

define _X_FILES /usr/openwin/lib/* /usr/openwin/share/*\\
/usr/openwin/bin/*

define _NETWORK_READ_FILES /etc/netconfig /etc/nsswitch.conf\\
/etc/.name_service_door

define _NETWORK_WRITE_FILES /dev/tcp /dev/udp /dev/ticotsord\\
/dev/ticlts, /dev/ticots

define _NETWORK_LIBS /usr/lib/libsocket.so.1 /usr/lib/libnsl.so\\
/usr/lib/nss_compat.so.1
```

Figure 8: A common specification file for Solaris 5.6.

```
# sandbox  spec for the browser class
# the browser sandbox takes three arguments -- the home directory
# the hosts it is allowed to connect to and the port(s)
# it is allowed to connect to.
params HOMEDIR HOSTSPEC PORTSPEC

# set up the env variables
putenv PATH=$HOMEDIR
putenv TERM=$_COMMON_TERM
putenv LD_LIBRARY_PATH=$_COMMON_LD_LIBRARY_PATH:$_NETWORK_LIBS
putenv DISPLAY=$_COMMON_DISPLAY

# _COMMON_READ and _COMMON_LIBS are accessible to all apps
path allow read $_COMMON_READ $_COMMON_LIBS $HOMEDIR
# browsers are allowed to read network config files and libs
path allow read $_NETWORK_READ_FILES $_NETWORK_LIBS
# browsers are allowed to read X data files and libs
path allow read $_X_FILES

# _COMMON_WRITE can be written by all (in this case /dev/zero)
# browsers are allowed to write HOMEDIR
path allow write $_COMMON_WRITE $HOMEDIR
# browsers are allowed to write networking device files
path allow write $_NETWORK_WRITE_FILES

# browsers are allowed to connect to all hosts in the argument
connect allow tcp $HOSTSPEC:$PORTSPEC
# broswers are allowed to connect to the X server
connect allow display

# all exec'ed children of browsers must be viewers
childbox viewer

# browsers are not allowed to access /etc/passwd
rename /etc/passwd /tmp/dummy
```

Figure 9: Sandbox example

A Secure Java™ Virtual Machine

Leendert van Doorn
leendert@watson.ibm.com
Global Security Analysis Laboratory
IBM T.J. Watson Research Center
Yorktown, NY

Abstract

The Java™ Virtual Machine is viewed by many as inherently insecure despite all the efforts to improve its security. In this paper we take a different approach to Java security and describe the design and implementation of a system that provides operating system style protection for Java code. We use hardware protection domains to separate Java classes, provide access control on cross domain method invocations, efficient data sharing between protection domains, and memory and CPU resource control. These security measures, when they do not violate the policy, are all transparent to the Java programs, even when a subclass is in one domain and its superclass is in another. To reduce the performance impact we group classes and share them between protection domains and map data on demand as it is being shared.

1. Introduction

Java™ [21] is a general-purpose programming language that has gained popularity as the programming language of choice for mobile computing. The language is used for World Wide Web programming [2], smart card programming [22], embedded device programming [16], and even for providing executable content for active networks [42]. Three reasons for this popularity are Java's portability, its security properties, and its lack of explicit memory deallocation.

Java programs are compiled into an intermediate representation called byte codes and run on a Java Virtual Machine (JVM). This JVM contains a byte code verifier that is essential for Java's security. Before execution begins the verifier asserts that the byte codes do not interfere with the execution of other programs by assuring it uses valid references and control transfers. Byte codes that successfully pass this verification are executed but still subject to number of other security measures implemented in the Java runtime system.

All of Java's security mechanisms depend on the correct implementation of the byte code verifier. In our opinion this is a flawed assumption and past experience has shown a number of security problems with this approach [11, 17, 35]. More fundamental is that from software engineering research it is known that every 1000 lines of code contain 35-80 bugs [7]. Even very thoroughly tested programs still contain on average about 0.5-3 bugs per 1000 lines of code [30]. Given that JDK 2 contains ~1.6M lines of code it is reasonable to expect 56K to 128K bugs. Granted, not all of these bugs are in security critical code but all code is security sensitive since it runs within a single protection domain.

Other unsolved security problems with current JVM designs are its vulnerability to denial of service attacks and its discretionary access control mechanisms. Denial of service attacks are possible because the JVM lacks proper support to bound the amount of memory and CPU cycles used by an application. The discretionary access control model is not always the most appropriate one for executing untrusted mobile code.

Interestingly, exactly the same security problems occur in operating systems. There they are solved by introducing hardware separation between different protection domains and controlled access between them. This hardware separation is provided by the memory management unit (MMU), an independent hardware component that controls all accesses to main memory. To control the resources used by a process an operating system limits the amount of memory it can use, assigns priorities to bias its scheduling, and can enforce mandatory access control. However, unlike programming language elements, processes are coarse grained and have primitive sharing and communication mechanisms.

An obvious solution to Java's security problems is to integrate the JVM with the operating system's process protection mechanisms. How to adapt the JVM efficiently and transparently (*i.e.,* such that multiple

Java applets can run on the same JVM while protected by the MMU) is a non-obvious problem. It requires a number of hard operating system problems to be resolved. These problems include: uniform object naming, object sharing, remote method invocation, thread migration, and protection domain and memory management.

The central goal of our work was the efficient integration of operating system protection mechanisms with a Java runtime system to provide stronger security guarantees. A subgoal was to be transparent with respect to Java programs. Where security and transparency conflicted they were resolved by a separate security policy. Using the techniques described in this paper we have build a prototype JVM that contains the following features:

- The transparent hardware assisted separation of Java classes, provided that they do not violate a preset security policy.
- The control over memory and CPU resources used by a Java class.
- The enforcement of mandatory access control for Java method invocations, class inheritance, and system resources.
- The employment of the *least privilege* concept and the introduction of a *minimal trusted computing base* (TCB).
- The JVM does not depend on the correctness of the Java byte code verifier for inter-domain protection.

In our opinion, a JVM using these techniques is much more amenable to an ITSEC or a Common Criteria evaluation than a pure software protection based system.

Our JVM consists of a small trusted component, called the *Java Nucleus*, which acts as a reference monitor and manages and mediates access between different protection domains (see figure 1). These protection domains contain one of more Java classes and their object instances. References to objects are capabilities [12] which are managed by the Java Nucleus.

For an efficient implementation of our JVM we depend on low-level operating system the functionality provided by *Paramecium* [41], an extensible operating system. The Java Nucleus uses its low-level protection domain and memory management facilities and its IPC for cross domain method invocations. The data is shared on demand using virtual memory remapping. When the data contains pointers to other data elements they are transparently shared as well. The garbage collector, which is a part of the Java Nucleus, handles

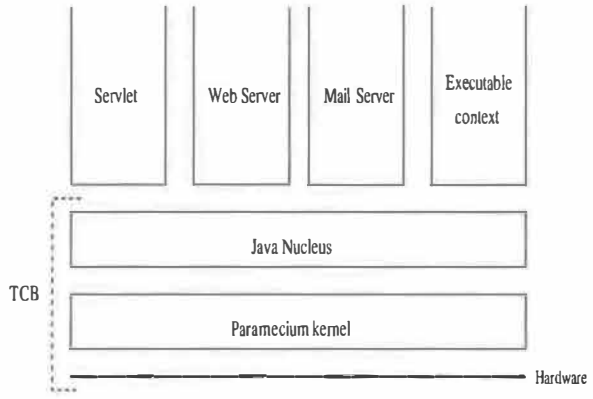


Figure 1. Secure JVM overview.

runtime data relocation, sharing and revocation of data elements, protection, and the reclaiming of unused memory cells over multiple protection domains.

In the next section of this paper we will describe the problems involved in language and operating system integration. Section 3 discusses the separation of concerns when designing a JVM architecture with a minimal TCB. It focuses on the security guarantees offered at run time and the corresponding threat model. Since our system relies on Paramecium it is described separately in section 4. Section 5 describes some of the key implementation details of our JVM. It discusses the memory model used by our JVM, its IPC mechanism, its data sharing techniques, and its garbage collector. Section 6 discusses some early experiences with our JVM, including a performance analysis and some example applications. Section 7 discusses related work and is followed by our conclusions in section 8.

2. Operating and Run Time System Integration

Integration of an operating system and a language runtime system has a long history (*e.g.*, Mesa/Cedar [39], Lisp Machines [29], Oberon [43], JavaOS [32], *etc.*), but none of these systems use hardware protection to supplement the protection provided by the programming language. In fact, most of these systems provide no protection or depend on a trusted code generator. For example, the Burroughs B5000 [9] enforced protection through a trusted compiler. It did not provide an assembler since it could be used to circumvent this protection.

Over the years these integrated systems have lost popularity in favor of time-shared systems with a process protection model. These systems provide better security and fault isolation by using hardware

separation between untrusted processes and controlling the communication between them. A side effect of this separation is that sharing is much harder and more inefficient.

The primary reasons why the transparent integration of a process protection model and a programming language are difficult are summarized in table 1. The key problem is their lack of a common naming scheme. In a process model each process has its own virtual address space, requiring techniques like pointer swizzling to translate addresses between different domains. Aside from the naming issues, the sharing granularity is different. Processes can share coarse grained pages while programs share many small variables. Reconciling the two as in distributed shared memory systems [27] leads to the undesirable effects of false sharing or fragmentation. Another distinction is the unit of protection. For a process this is an protection domain, for programs it is a module, class, object, *etc.* Finally, processes use rudimentary IPC facilities to communicate that can send and receive blocks of data. Programs on the other hand use procedure calls and memory references.

In order to integrate a process protection model and a programming language we need to adapt some of the key process abstractions. Adapting them is hard to do in a traditional operating system because they are hardwired into the system. Extensible operating systems on the other hand provide much more flexibility (*e.g.,* Paramecium, OSKit [19], L4/LavaOS [28], ExOS [15], and SPIN [6]). For example, in our system the Java Nucleus acts as a special purpose kernel for Java programs. It controls the protection domains that contain Java programs, creates memory mappings, handles all protection faults for these domains and controls cross protection domain invocations. These functions are hard to implement on a traditional system but straightforward on an extensible operating system. A second enabling feature of extensible operating systems is the dramatic improvement in cross domain transfer cost by eliminating unnecessary abstractions [5, 24, 28, 33]. This makes the tight integration of multiple protection domains feasible. Another advantage of using an extensible kernel is that they tend to be several orders smaller than traditional kernels. This is a desirable property since the kernel is part of the TCB.

	Process Protection Model	**Programming Language**
Name space	disjoint	single
Granularity	pages	variables
Unit	protection domain	class/object
Communication	IPC	call/memory

Table 1. Process protection model *vs.* programming language.

For a programming language to benefit from hardware separation it has to exhibit a number of requirements. The first one is that the language must contain a notion of a unit of protection. These units form the basis of the protection system. Examples of these units are classes, objects, and modules. Each of these units must have one or more interfaces to communicate with other units. Furthermore there needs to be non-language reasons to separate these units, like running multiple untrusted applets simultaneously on the same system. The last requirement is that the language needs to use a typed garbage collection system rather than programmer managed dynamic memory. This requirement allows a third party to manage, share and relocate the memory used by a program.

In this paper we concentrate on the integration of Paramecium and Java. While both fulfill the requirements listed above, the techniques are applicable to other operating systems and programming languages.

3. Separation of Concerns

The goal of our secure JVM is to minimize the trusted computing base (TCB) for a Java run-time system. For this it is important to separate security concerns from language protection concerns and establish what type of security enforcement has to be done at compile time, loading time, and run time.

At compile time the language syntax and semantic rules are enforced by a compiler. This enforcement ensures valid input for the transformation process of source code into byte codes. Since the compiler is not trusted the resulting byte codes cannot be trusted and therefore we can not depend on the compiler for security enforcement.

At load time a traditional JVM loads the byte codes and relies on the byte code verifier and various run-time checks to enforce the Java security guarantees. As discussed in the introduction, we do not rely on the Java byte code verifier for security for its size, complexity, and track record. Instead we aim at minimizing the TCB and use hardware fault isolation between groups of classes and their object instances and control access to methods and state shared between them. A separate security policy defines which classes are grouped together in a single protection domain and which methods they may invoke on different protection domains. It is important to realize that all classes

within a single protection domain have the same trust level. Our system provides strong protection guarantees between different protection domains, *i.e.*, inter-domain protection. It does not enforce intra-domain protection, this is left to the run-time system if desirable. This does not constitute a breakdown of security of the system. It is the policy that defines the security. If two classes that are in the same domain, *i.e.*, have the same trust level, misbehave with respect to one another this clearly constitutes a failure in the policy specification. These two classes should not have been in the same protection domain.

The run-time security provided by our JVM consists of hardware fault isolation among groups of classes and their object instances by isolating them into multiple protection domains and controlling access to methods and state shared between them. Each security policy, a collection of permissions and accessible system resources, defines a protection domain. All classes with the same security policy are grouped into the same domain and have unrestricted access to the methods and state within it. Invocations of methods in other domains pass through the Java Nucleus. The Java Nucleus is a trusted component of the system and enforces access control based on the security policy associated with the source and target domain.

The Java Nucleus consists of four components: a class loader, a garbage collector, a thread system, and an IPC component. The class loader loads a new class, translates the byte codes into native machine codes, and deposits them into a specified protection domain. The garbage collector allocates and collects memory over multiple protection domains, assists in sharing memory among them, and implements memory resource control. The thread system provides the Java threads of control and maps them directly onto Paramecium threads. The IPC component implements cross protection domain invocations, access control, and CPU resource usage control.

The JVM's trust model (*i.e.*, what is included in the minimal trusted computing base) depends on the correct functioning of the garbage collector, IPC component, and thread system. We do not depend on the correctness of the byte code translator. When the byte code translator is trusted to separate executable content from data certain optimizations are possible. These are described in section 5.2.

References to memory cells (primitive types or objects) act as capabilities [12] and can be passed to other protection domains as part of a cross domain method invocation (XMI) or object instance state sharing. Passing an object reference results in passing the full closure of the reference. That is, all cells that can be obtained by dereferencing the pointers that are contained in the cell of which the reference is passed. Capabilities can be used to implement the notion of least privilege but suffer from the classical confinement and revocation problem. Solving these is straightforward since the Java Nucleus acts as a reference monitor. However, this violates the Java language transparency requirement (see section 8).

Our system does not depend on the Java security features such as byte code verification, discretionary access control through the security manager, and its type system. We view these as language security measures that assist the programmer during program development and they should not be confused or combined with system security measures. The latter isolates and mediates access between protection domains and resources; these measures are independent of the language. However, integrating an operating system style protection with the semantic information provided by the language runtime system does allow finer grained protection and sharing than is possible in contemporary systems.

The security provided by our JVM is defined in a security policy. The elements that comprise this policy are listed in table 2. They consist of a set of system resources available to each protection domain, classes whose implementation is shared between multiple domains, object instance state that is shared, and access control for each cross domain method invocation.

The first policy element is a per method access control for cross protection domain invocations. Each method has associated with it a list of domains that can invoke it. If the invocation target is not in this set, access is denied. Protection is between domains, not within domains, hence there is no access control for method invocations within the same domain.

To reduce the number of cross protection domain calls (XMIs) the class text (instructions) can be shared between multiple domains. This is analogous to text sharing in UNIX, where the instructions are loaded into

Granularity	Mechanism
Method	Invocation access control
Class	Instruction text sharing between domains
Class	Object sharing between domains
Reference	Opaque object handle
System	Paramecium name space per domain

Table 2. Security policy elements.

memory only once and mapped into each domain that uses it. This reduces memory requirements. In our case it eliminates the need for expensive XMIs. The object instance state is still private to each domain.

Object instance state is transparently shared between domains when references to it are passed over XMIs or when an object inherits from a class in a different protection domain. Which objects can be passed between domains is controlled by the Java programs and not by the JVM. Specifying this as part of the security policy would break the Java language transparency requirement. Per-method access control gives the JVM the opportunity to indirectly control which references are passed.

In circumstances where object instance state sharing is not desirable a class can be marked as non-sharable for a specified domain. Object references of this class can still be passed to the domain but cannot be dereferenced by it. This situation is similar to client/server mechanisms where the reference acts as an opaque object handle. Since Java is not a real object-oriented language, it allows clients to directly access object state, this mechanism is not transparent for some Java programs.

Fine grained access control over the system resources is provided by the Paramecium name space mechanism. If a service name is not in the name space of a protection domain, that domain cannot get access to the service. The name space for each protection domain is constructed and controlled by our Java Nucleus.

To reduce the number of XMIs the classes with the same security policy are grouped into the same protection domain. The number of XMIs can be further reduced by sharing the instruction text of class implementations between different domains.

Table 3 summarizes the potential threats our JVM can handle, together with their primary protection mechanism. Some threats, such as covert channels, are not handled in our system. Other threats, such as denial of service attacks caused by improper locking behavior are considered policy errors. The offending applet should not have been given access to the mutex.

Threat	Protection mechanism
Fault isolation	Protection domains
Denial of service	Resource control
Forged object references	Garbage collector
Illegal object invocations	XMI access control

Table 3. Threat model.

4. Paramecium

Paramecium [41] is an extensible object-based operating system for building application-specific operating systems. It consist of a protected and trusted nanokernel which implements only those services that cannot be moved into the application protection domain without jeopardizing the system's integrity. All other system components, like thread packages, device drivers, and virtual memory implementations reside outside this nucleus.

The kernel provides three basic services which all use a protection domain as their unit of granularity. These services are: event management, memory management, and name space management. Each resource managed by these services is identified by a capability.

The first basic service provided by the kernel is event management. Paramecium uses preemptive events for handling interrupts, traps, and interprocess communication. Associated with each event is a handler which is executed when the event is raised. A handler consists of a protection domain identifier, the address of a call-back function, and a stack pointer. Raising an event causes control to be transfered to the handler specified by the protection domain identifier and call-back function using the specified handler stack. The event service also has provisions for the handler to determine the caller's domain.

The second basic service provided by the kernel is memory management. This service manages physical and virtual memory. The physical memory service allocates physical pages which are then mapped onto a virtual memory address using the virtual memory service. Each physical page is identified by a generic system-wide resource identifier. Shared memory is implemented by passing this physical page resource identifier to another protection domain and having it map it into its virtual address space.

Paramecium implements multiple protection domains. Each protection domain is a mapping of virtual to physical pages together with a set of domain specific events. These domain events are raised on, for example, division by zero traps when this particular domain is in control.

The protection domain's virtual memory space is managed by the virtual memory service. This service implements functions to map physical pages onto virtual addresses, set virtual page attributes (*e.g.*, read-only, read-write, execute-only), and unmap them. Each

virtual page has associated with it a fault event that is raised, if set, when a fault occurs on an address within that page. These faults include among others: an instruction access fault when a page is marked as non-executable or a data access fault when a write occurs to a read-only page. A generic fall back event is raised when no event is associated with a virtual page.

Event handlers for a virtual page fault do not need to reside in the same protection domain as the one where the fault occurs. These can be handled by an external process that might, for example, implement demand paging or, as in our secure JVM, maintain full control over the protection domains that are executing Java programs.

The last basic service provided by the kernel is name space management. Paramecium organizes its components into objects and interface references to instantiated objects. These are stored in a hierarchical name space. Each protection domain has a view of its own subtree of the name space, the kernel address space has a view of the entire tree including all the subtrees of different protection domains (see figure 2).

Standard operations exist to bind to an existing object reference, to load an object from the repository, and to obtain an interface from a given object reference. Binding to an object happens at runtime. One would reconfigure a particular service by overriding its name. A search path mechanism exists to control groups of overrides. When an object is owned by a different protection domain the name service automatically instantiates proxy interfaces.

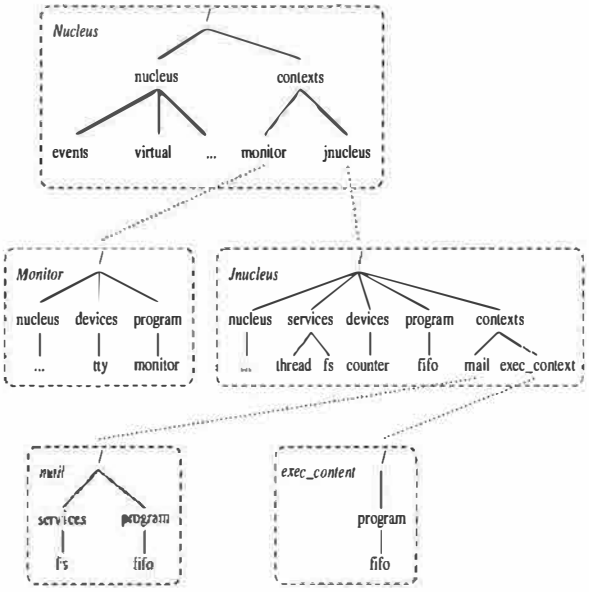


Figure 2. Paramecium name spaces.

When a protection domain is created it is passed the root of its name space. Depending on the names in its space, it can contact external services. For example, the file server is known as "/services/fs". Binding to this name and obtaining a file system interface enables the process to create and delete files by invoking the methods from the interface. When the name is not present in the name space no file system operations are possible. By default protection domains are created with an empty name space.

Applications that use this kernel are implemented as and constructed from separate components. One example of a component is Paramecium's thread package. This package provides a priority scheduler and support for migratory threads [18]. Migratory threads can migrate from one protection domain to another without actually switching threads or changing the thread identifier. This saves a number of context switches which are required by systems that hand off work from a thread in one domain to a thread in another domain.

5. Secure Java Virtual Machine

The Java Nucleus forms the minimal trusted computing base (TCB) of our secure JVM. This section describes the key techniques and algorithms used by the Java Nucleus.

In short, the Java Nucleus provides a uniform naming scheme for all protection domains, including the Java Nucleus. It provides a single virtual address space where each protection domain can have a different protection view. All cross protection domain method invocations (XMIs) pass through our Java Nucleus which controls access, CPU and memory resources. Data is shared on demand between multiple protection domains, *i.e.* whenever a reference to shared data is dereferenced. Our Java Nucleus uses shared memory and runtime reallocation techniques to accomplish this. Only references passed over an XMI or object instances whose inherited classes are in different protection domains can be accessed, others will cause security violations. These protection mechanisms depend on our garbage collector to allocate and deallocate typed memory, relocate memory, control memory usage, and keep track of ownership and sharing status.

It is possible to use the techniques described below to build a secure JVM using an interpreter rather than a compiler. Each protection domain would then have a shared copy of the interpreter interpreting the Java byte codes for that protection domain. We have

not explored such an implementation because of the obvious performance benefits of executing generated machine code.

The next subsections describe the key techniques and algorithms in greater detail.

5.1. Memory Organization

The Java Virtual Machine assumes a single address space in which references can be passed between method invocations. This and Java's dependence of garbage collection dictated our memory organization.

Inspired by single address space operating systems [10], we organized memory into a single virtual address space. Each protection domain has, depending on its privileges, a view onto this address space. This view includes a set of physical memory pages to virtual mappings together with their corresponding access rights. A small portion of the virtual address space is reserved by each protection domain to store per domain specific data.

Central to the protection domain scheme is the Java Nucleus (see figure 3). The Java Nucleus is analogous to an operating system kernel. It manages a number of protection domains and has full access to all memory mapped into these domains and their corresponding access permissions. The protection domains themselves cannot manipulate the memory mappings or the access rights of their virtual memory pages. The Java Nucleus handles both data and instruction access (*i.e.*, page) faults for these domains. Page faults are turned into appropriate Java exceptions when they are not used by the system.

For convenience all the memory available to all protection domains is mapped into the Java Nucleus with read/write permission. This allows it to quickly access the data in different protection domains. Because memory addresses are unique and the memory pages are mapped into the Java Nucleus protection domain, the Java Nucleus does not have to map or copy memory as an ordinary operating system kernel.

The view different protection domains have of the address space depends on the mappings created by the Java Nucleus. Consider figure 3. A mail reader application resides in the context named mail. For efficiency reasons, all classes constituting this application reside in the same protection domain; all executable content embedded in an e-mail message is executed in a separate domain, say executable content. In this example memory region 0 is mapped into the context executable content. Part of this memory contains

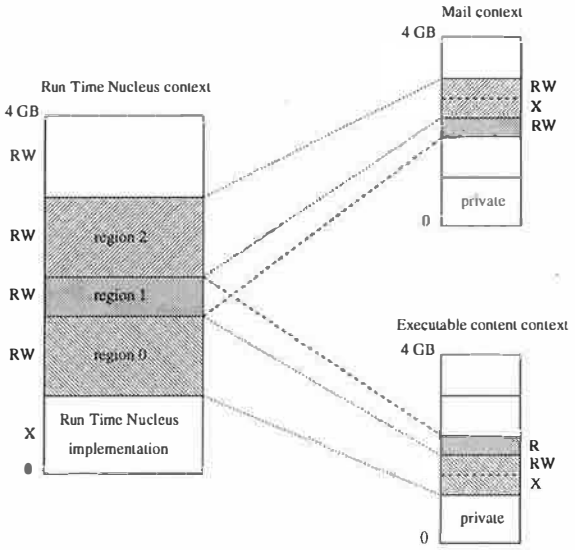


Figure 3. Java nucleus memory map.

executable code and has the execute privilege associated with it. Another part contains the stack and data and has the read/write privilege. Region 0 is only visible to the executable content context and not to the mail context. Likewise, region 2 is not visible to the executable content context. Because of the hardware memory mappings these two contexts are physically separated.

Region 1 is used to transfer data between the two contexts and is set up by the Java Nucleus. Both contexts have access to the data, although the executable content context has only read access. Violating this access privilege causes a page (data access) fault to be generated which is handled by the Java Nucleus. It will turn the fault into a Java exception.

5.2. Cross Domain Method Invocations

A cross domain method invocation (XMI) mimics a local method invocation except that it crosses a protection domain boundary. A vast amount of literature exists on low latency cross domain control transfer [5, 24, 28]. Our XMI mechanism is loosely based on Paramecium's system call mechanism which uses events. The following example illustrates the steps involved in an XMI.

Consider the protection domains A and B and a method M which resides in domain B. When A calls method M an instruction fault will occur since M is not mapped into context A. The fault causes an event to be raised in the Java Nucleus. The event handler for this fault is passed two arguments: the fault address (*i.e.*,

method address) and the fault location (*i.e.*, call instruction). Using the method address, the Java Nucleus determines the method information which contains the destination domain and the access control information. Paramecium's event interface is used to determine the caller domain. Based on this information, an access decision is made. If access is denied, a security exception is raised in the caller domain.

Using the fact that method information is static and that domain information is static for code that is not shared, we can improve the access control check process. Rather than looking up this information, the Java Nucleus stores a pointer to it in the native code segment of the calling domain. The information can then be accessed quickly, using a fixed offset and fault location parameter. Method calls are achieved through special trampoline code that embeds these two values. More precisely, the call trampoline code fragment in context A for calling method M appears as (in SPARC [38] assembly):

```
    call  M                ! call method M
     mov   %g0, %i0        ! nil object
    b,a   next_instr       ! branch over
    .long <caller domain>  !   JNucleus data
    .long <method info>    !   JNucleus data
next_instr:
```

The information stored in the caller domain must be protected from tampering. This is achieved by mapping all executable native code as execute only; only the Java Nucleus has full access to it.

When access is granted for an XMI, an event is associated with the method if one is not already present. Then the arguments are copied into the registers and onto the event handler stack as dictated by the calling frame convention. No additional marshalling of the parameters is required. Both value and reference parameters are passed unchanged. Using the method's type signature to identify reference parameters, we mark data references as exported roots (*i.e.*, garbage collection roots). Instance data is mapped on demand as described in the next section. Invoking a method on an object reference causes an XMI to the method implementation in the object owner's protection domain.

Virtual method invocations, where a set of specific targets is known at compile-time but the actual target only at runtime, require a lookup in a switch table. The destinations in this table refer to call trampolines rather than the actual method address. Each call trampoline consists of the code fragment described above.

Using migratory threads, an XMI extends the invocation chain of the executing thread into another protection domain. Before raising the event to invoke the method, the Java Nucleus adjusts the thread priority according to the priority of the destination protection domain. The original thread priority is restored on the method return. Setting the thread priority enables the Java Nucleus to control the CPU resources used by the destination protection domain.

Local method invocations use the same method call trampoline as the one outlined above, except that the Java Nucleus does not intervene. This is because the method address is available locally and does not generate a fault. The uniform trampoline allows the Java Nucleus to share class implementations among multiple protection domains by mapping them in. For example, simple classes like the java.lang.String or java.lang.Long can be shared by all protection domains without security implications. Sharing class implementations reduces memory use and improves performance by eliminating XMIs. XMIs made from a shared class do not have their caller domain set, since there can be many caller domains, and require the Java Nucleus to use the system authentication interface to determine the caller.

5.3. Data Sharing

Passing parameters, as part of a cross domain method invocation (XMI), requires little more than copying them by value and marking the reference variables as exported roots. Subsequent accesses to these references will cause a protection fault unless the reference is already mapped in. The Java Nucleus, which handles the access fault, will determine whether the faulting domain is allowed access to the variable referenced. If allowed, it will share the page on which the variable is located.

Sharing memory on a page basis traditionally leads to false sharing or fragmentation. Both are clearly undesirable. False sharing occurs when a variable on a page is mapped into two address spaces and the same page contains other unrelated variables. This clearly violates the confinement guarantee of the protection domain. Allocating each variable on a separate page results in fragmentation with large amounts of unused physical memory. To share data efficiently between different address spaces, we use the garbage collector to reallocate the data at runtime. This prevents false sharing and fragmentation.

Consider figure 4 which shows the remapping process to share a variable *a* between the mail context and the executable content context. In order to relocate this variable we use the garbage collector to update all the references. To prevent race conditions the threads within or entering the contexts that hold a reference to

a are suspended (step 1). Then the data, a, is copied onto a new memory page (or pages depending on its size) and referred to as a'. The other data on the page is not copied, so there is no risk of false sharing. The garbage collector is then used to update all references that point to a into references that point to a' (step 2). The page holding a' is then mapped into the other context (step 3) Finally, the threads are resumed, and new threads are allowed to enter the unblocked protection domains (step 4). The garbage collector will eventually delete a since it does not have any references to it.

Other variables that are shared between the same protection domains are tagged onto the already shared pages to reduce memory fragmentation. The process outlined above can be applied recursively. That is, when a third protection domain needs access to a shared variable the variable is reallocated on a page that is shared between the three domains.

In order for the garbage collector (see section 5.4) to update the cell references it has to be exact. That is, it must keep track of the cell types and of references to each cell to distinguish valid pointers from random integer values. The updating itself can either be done by a full walk over all the in-use memory cells or by arranging each cell to keep track of the objects that reference it. The overhead of the relocation is amortized over subsequent uses.

Besides remapping dynamic memory, the mechanism can also be used to remap static (or class) data. Absolute data memory references can occur within the native code generated by the just-in-time compiler. Rather than updating the native code on each relocation, the just-in-time compiler generates an extra indirection to a placeholder holding the actual reference.

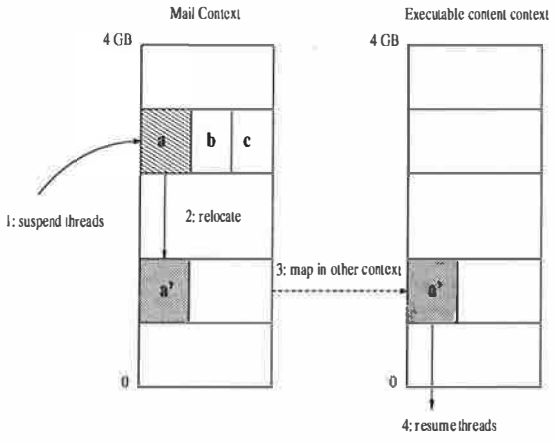


Figure 4. Data remapping between address spaces.

This placeholder is registered with the garbage collector as a reference location.

Data remapping is not only used to share references passed as parameters over an XMI, but also to share object instance data between sub and superclasses in different protection domains. Normally, object instances reside in the protection domain in which their class was loaded. Method invocations on that object from different protection domains are turned into XMIs. In the case of an extended (*i.e.*, inherited) class the object instance state is shared between the two protection domains. This allows the sub and superclass methods to directly access the instance state rather than capturing all these accesses and turning them into XMIs. To accomplish this our JVM uses the memory remapping technique outlined above.

The decision to share object instance state is made at the construction time of the object. Construction involves calling the constructor for the class followed by the constructors for its parent classes. When the parent class is in a different protection domain the constructor invocation is turned into an XMI. The Java Nucleus performs the normal access control checks as for any other XMI from a different protection domain. The object instance state, that is passed implicitly as the first argument to the constructor call, is marked as an exportable root. The mechanisms involved in marking memory as an exportable root are discussed in the next section.

Java uses visibility rules (*i.e.*, *public* and *protected*) to control access to parts of the object instance state. Enforcing these rules through memory protection is straightforward. Each object's instance state is partitioned into a shared and non-shared part. Only the shared state can be mapped.

An example of state sharing between super and subclass is shown in figure 5. Here the class BitMap and all its instances reside in protection domain A. Protection domain B contains all the instances of the class Draw. This class is an extension of the BitMap class which resides in a different protection domain. When a new instance of Draw is created the Draw constructor is called to initialize the class. In this case the constructor for Draw is empty and the constructor for the superclass BitMap is invoked. Invoking this constructor will cause a transfer into the Java Nucleus.

The Java Nucleus first checks the access permission for domain B to invoke the BitMap constructor in domain A. If granted, the object pointer is marked as an exportable root and passed as the first implicit parameter. Possible other arguments are copied as part of the XMI mechanism and the remote invocation is

```
class BitMap { // Domain A
    private static int N = 8, M = 8;
    protected byte bitmap[][];

    protected BitMap() {
        bitmap = new byte[N/8][M];
    }

    protected void set(int x, int y) {
        bitmap[x/8][y] |= 1<<(x%8);
    }
}

class Draw extends BitMap { // Domain B
    public void point(int x, int y) {
        super.set(x, y);
    }

    public void box(int x1, int y1,
                    int x2, int y2) {
        for (int x = x1; x < x2; x++)
            for (int y = y1; y < y2; y++)
                bitmap[x/8][y] |= 1<<(x%8);
    }
}
```

Figure 5. Simple box drawing class.

performed. The BitMap constructor then assigns a new array to the bitmap field in the Draw object. Since the assignment is the first dereference for the object it will be remapped into domain A. When the creator of the Draw object calls box and dereferences bitmap it will be remapped into domain B (because the array is reachable from an exported root cell to domain A; see next section). Further calls to box do not require this remapping. A call to point results in an XMI to domain A where the superclass implementation resides. Since the Draw object was already remapped by the constructor it is does not require any remapping.

Whenever a reference is shared among address spaces all references that are reachable from it are also shared and will be mapped on demand when referred to. This provides full transparency for Java programs which assume that a reference can be passed among all its classes. A potential problem with on-demand remapping is that it dilutes the programmers' notion of what is being shared over the life-time of a reference. This might obscure the security of the system. To strengthen the security, an implementation might decide not to support remapping of objects at all or provide a proactive form of instance state sharing. Not supporting instance state sharing prevents programs that use the object oriented programming model from being separated into multiple protection domains. For example, it precludes the isolation and sharing of the AWT package in a separate protection domain.

The implementation has to be conservative with respect to references passed as arguments to cross domain method invocations and has to unmap them whenever possible to restrict their shared access. Rather than unmapping at the invocation return time, which would incur a high call overhead, we defer this until garbage collection time. The garbage collector is aware of shared pages and determines whether they are reachable in the context they are mapped in. If they are unreachable, rather than removing all the bookkeeping information the page is marked invalid so it can be remapped quickly when it is used again. This does not work very well if the page contains two variables of which only one is passed by reference. In that case a new remapping is required. To reduce the amount of remapping, separate pages are used for shared instance state and state passed as a reference to a cross domain invocation.

5.4. Garbage Collection

Java uses garbage collection [25] to reclaim unused dynamic memory. In our design we use a non-collecting incremental traced garbage collector which is part of the Java Nucleus. The garbage collector is responsible for collecting memory in all the address spaces it manages. A centralized garbage collector has the advantage that it is easier to share memory between different protection domains and to enforce central access and resource control. An incremental garbage collector has better real time properties than non-incremental collectors.

More precisely, the garbage collector for our secure Java machine must have the following properties:

(1) Collect memory over multiple protection domains and protect the bookkeeping information from the potentially hostile domains.

(2) Relocate data items at runtime. This property is necessary for sharing data across protection domains. Hence, we use an exact garbage collector rather than a conservative collector [8].

(3) Determine whether a reference is reachable from an exported root. Only those variables that can be obtained via a reference passed as an XMI argument or instance state are shared.

(4) Maintain, per protection domain, multiple memory pools with different access attributes. These are execute only, read-only, and read-write pools that contain native code, read-only and read-write data segments respectively.

(5) Enforce resource control limitations per protection domain.

As discussed in the previous section all protection domains share the same virtual address map albeit with different protection views of it. The Java Nucleus protection domain, which contains the garbage collector, has full read-write access to all available memory. Hence the ability to collect memory over different domains is confined to the Java Nucleus.

A key feature of our garbage collector is that it integrates collection and protection. Classical tracing garbage collection algorithms assume a single address space in which all memory cells have the same access rights. In our system cells have different access rights depending on the protection domain accessing it and cells can be shared among multiple domains. Although access control is enforced through the memory protection hardware, it is the garbage collector that has to create and destroy the memory mappings.

The algorithm we use (see the pseudo-code in figure 6) is an extension of a classic mark-sweep algorithm which runs concurrently with the mutators [13]. The original algorithm uses a tricolor abstraction in which all cells are painted with one of the following colors: *black* indicates that the cell and its immediate descendents have been visited and are in use; *grey* indicates that the cell has been visited but not all of its descendents, or that its connectivity to the graph has changed; and *white* indicates untraced (*i.e.*, free) cells. The garbage collection phase starts with all cells colored white and terminates when all traceable cells have been painted black. The remaining white cells are free and can be reclaimed.

To extend this algorithm to multiple protection domains we associate with each cell its owner domain and an export set. An export set denotes to which domains the cell has been properly exported. Garbage collection is performed on one protection domain at a time, each keeping its own color status to assist the marking phase. The marking phase starts by coloring all the root and exported root cells for that domain as grey. It then continues to examine all cells within that domain. If one of them is grey it is painted black and all its children are marked grey until there are no grey cells left. After the marking phase, all cells that are used by that domain are painted black. The virtual pages belonging to all the unused white cells are unmapped for that domain. When the cell is no longer used in any domain it is marked free and its storage space is reclaimed. Note that the algorithm in figure 6 is a simplification of the actual implementation, many improvements (such as [14, 26, 36]) are possible. A correctness proof of the algorithm follows from [13].

```
COLLECT():
    for (;;) {
      for (d in Domains)
        MARK(d)
      SWEEP();
    }

MARK(d: Domain): // marker phase
    color[d, (exported) root set] = grey
    do {
      dirty = false
      for (c in Cells) {
        if (color[d, c] == grey) {
          color[d, c] = black
          for (h in children[c]) {
            color[d,h] = grey
            if (EXPORTABLE(c, h))
              export[d,h] |= export[d,c]
          }
          dirty = true
        }
      }
    } while (dirty)

SWEEP(): // sweeper phase
    for (c in Cells) {
      used = false
      for (d in Domains) {
        if (color[d, c] == white) {
          export[d, c] = nil
          UNMAP(d, c)
        } else
          used = true
        color[d, c] = white
      }
      if (used == false)
        DELETE(c)
    }

ASSIGN(a, c): // pointer assignment
    *a = c
    d = current domain
    export[d,c] |= export[d,a]
    if (color[d, c] == white)
      color[d, c] = grey

EXPORT(d: Domain, c: Cell): // export object
    color[d,c] = grey
    export[d,c] |= owner(c)
    export[owner(c),c] |= d
```

Figure 6. Multiple protection domain garbage collection.

Cells are shared between other protection domains by using the remapping technique described in the previous section. In order to determine whether a protection domain *d* has access to a cell *c* the Java Nucleus has to examine the following three cases: The trivial case is where the owner of *c* is *d*. In this case the cell is already mapped into domain *d*. In the second case the owner of *c* has explicitly given access to *d* as part of an XMI parameter or instance state sharing or

is directly reachable from such an exported root. This is reflected in the export information kept by the owner of c. Domain d has also access to cell c if there exists a transitive closure from some exported root r owned by the owner of c to some domain z. From this domain z there must exist an explicit assignment which resulted in c being inserted into a data structure owned by d or an XMI from the domains z to d passing cell c as an argument. In the case of an assignment the data structure is reachable from some other previously exported root passed by d to z. To maintain this export relationship each protection domain maintains a private copy of the cell export set. This set, usually nil and only needed for shared memory cells, reflects the protection domain's view of who can access the cell. A cell's export set is updated on each XMI (*i.e.*,export) or assignment as shown in figure 6.

Some data structures exist prior to, for example, an XMI passing a reference to it. The export set information for these data structures is updated by the marker phase of the garbage collector. It advances the export set information from a parent to all its siblings taking the previously mentioned export constraints into account.

Maintaining an export set per domain is necessary to prevent forgeries. Consider a simpler design in which the marker phase advances the export set information to all siblings of a cell. This allows the following attack where an adversary forges a reference to an object in domain d and then invokes an XMI to d passing one of its data structures which embeds the forged pointer. The marker phase would then eventually mark the cell pointed to by the forged reference as exported to d. By maintaining for each cell a per protection domain export set forged pointers are impossible.

Another reason for keeping a per protection domain export set is to reduce the cost of a pointer assignment operation. Storing the export set in the Java Nucleus would require an expensive cross protection domain call for each pointer assignment, by keeping it in user space this can be eliminated. Besides, the export set is not the only field that needs to be updated. In the classic Dijkstra algorithm the cell's color information needs to be changed to *grey* on an assignment (see figure 6). Both these fields are therefore kept in user space.

The cell bookkeeping information consists of three parts (see figure 7). The public part contains the cell contents and its per domain color and export information. These parts are mapped into the user address space, where the color and export information is stored in the per domain private memory segment (see 5.1). The nucleus part is only visible to the Java Nucleus. A page contains one or more cells where for each cell the content is preceded by a header pointing to the public information. The private information is obtained by hashing the page frame number to get the per page information which contains the private cell data. The private cell data contains pointers to the public data for all protection domains that share this cell. When a cell is shared between two or more protection domains the pointer in the header of the cell refers to public cell information stored in the private domain specific portion of the virtual address space. The underlying physical pages in this space are different and private for each protection domain.

To amortize the cost of garbage collection, our implementation stores one or more cells per physical memory page. When all the cells are free the page is added to the free list. As stated earlier, each protection domain has three memory pools: an execute-only pool, a read-only pool, and a read-write pool. Cells are allocated from these pools depending on whether they contain executable code, constant data, or volatile data. When memory becomes really tight, pages are taken from their free lists, their virtual pages are unmapped, and their physical pages returned to the system physical page pool. This allows them to be re-used for other domains and pools.

Exposing the color and export set fields requires the garbage collector to be very conservative in handling these user accessible data items. It does not,

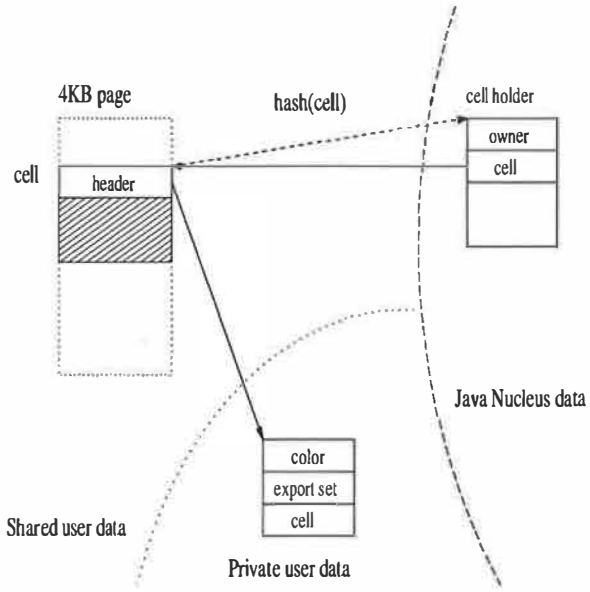


Figure 7. Garbage collection cell structure.

however, reduce the security of our system. The user application can, at most, cause the marker phase to loop forever, cause its own cells that are still in use to be deallocated, or hang on to shared pages. These problems can be addressed by bounding the marker loop phase by the number of in-use cells. Deleting cells that are in use will cause the program to fail eventually, and hanging on to shared pages is not different from the program holding on to the reference.

When access to a cell is revoked, for example as a result of an XMI return, its color is marked grey and it is removed from the receiving domain's export set. This will cause the garbage collector to reexamine the cell and unmap it during the sweep phase when there are no references to it from that particular domain.

To relocate a reference the Java Nucleus forces the garbage collector to start a mark phase and update the appropriate references. Since the garbage collector is exact it only updates actual object references. An alternative design for relocation is to add an extra indirection for all data accesses. This indirection eliminates the need for explicit pointer updates. Relocating a pointer consists of updating its entry in the table. This design, however, has the disadvantage that it imposes an additional cost on every data access rather than the less frequent pointer assignment operation and prohibits aggressive pointer optimizations by smart compilers.

The amount of memory per protection domain is constrained. When the amount of assigned memory is exhausted an appropriate exception is generated. This prevents protection domains from starving other domains of memory.

6. Experience

Our prototype implementation is based on Kaffe, a freely available JVM implementation [40]. We used its class library implementation and JIT compiler and we reimplemented the IPC, garbage collector, and thread subsystems. Our prototype implements multiple protection domains and data sharing. For convenience, the Java Nucleus contains the JIT compiler and all the native class implementations. It does not yet provide support for text sharing of class implementations and has a simplified security policy description language. Currently, the security policy defines protection domains by explicitly enumerating the classes that comprise it and the access permissions for each individual method. The current garbage collector is not exact for the evaluation stack and uses a weaker form to propagate export set information.

The trusted computing base (TCB) of our system is formed by the Paramecium kernel and the Java Nucleus. The size of our Paramecium kernel is about 11000 lines of commented header files, and C++/assembler code. The current Java Nucleus is about 22000 lines of commented header files and C++ code. This includes the JIT component, threads, and much of the Java run-time support. In a system that supports text sharing the Java Nucleus can be reduced considerably.

A typical application of our JVM is that of a web server written in Java that supports servlets, like W3C's JigSaw. Servlets are Java applets that run on the web server and extend the functionality of the server. They are activated in response to requests from a web browser and act mainly as a replacement for CGI scripts. Servlets run on behalf of a remote client and can be loaded from a remote location. They should therefore be kept isolated from the rest of the web server.

Our test servlet is the SnoopServlet that is part of the Sun's Java servlet development kit [37]. This servlet inherits from a superclass HttpServlet which provides a framework for handling HTTP requests and turning them into servlet method calls. The SnoopServlet implements the GET method and returns a web page containing a description of the browser capabilities. This page is served to the client by a simple web server which is implemented by the HttpServlet superclass. For our test the web server and all class libraries are loaded in protection domain WS, the servlet implementation is confined to Servlet.

The WS domain makes 2 calls into the Servlet domain, one to the constructor for SnoopServlet object and one to the doGet method implementing HTTP GET. This method has two arguments, the servlet request and reply objects. Invoking methods on these causes XMIs back into the WS domain. In this test a total of 217 XMIs occurred. Many of these calls are to runtime classes such as java/io/PrintWriter (62) and java.lang.StringBuffer (101). In an implementation that supports text sharing these calls would be local procedure calls and only 33 calls would require an actual XMI to the web server. Many of these XMIs are the result of queries from the servlet to the browser.

The number of objects that are shared and therefore relocated between the WS and Servlet domains are 47. Most of the relocated objects are static strings (45) which are used as arguments to print the browser information. These too can be eliminated by using text sharing since the underlying implementation of print uses a single buffer. In that case only a single buffer needs to be relocated. The remaining relocated objects are the

result of the the HttpServlet class keeping state information.

The cost of an XMI from the WS domain to the Servlet domain is about 11 μsec. This high cost can be purely attributed to the cost of a Paramecium's IPC on a 50 MHz SPARC, which is 5 μsec. The overhead for Java XMIs is negligible.

7. Related Work

Our system is the first to use hardware fault isolation on commodity components to supplement language protection by tightly integrating the operating system and language runtime system. In our design we concentrated on Java, but our techniques are applicable to other languages as well (*e.g.*, SmallTalk [20] and Modula3 [31]) provided they use garbage collection, have well defined interfaces, and distinguishable units of protection. A number of systems provide hardware fault isolation by dividing the program into multiple processes and use a proxy based system like RMI or CORBA, or a shared memory segment for communication between them. Examples of these systems are the J-Kernel [23] and cJVM [1]. This approach has a number of drawbacks that are not found in our system:

(1) Most proxy mechanisms use marshalling to copy the data. Marshalling provides copy semantics which are incompatible with the shared memory semantics required by the Java language.

(2) The overhead involved in marshalling and unmarshalling the data is significant compared to on demand sharing of data.

(3) Proxy techniques are based on interfaces and are not suited for other communication mechanisms such as instance state sharing. The latter is important for object oriented languages.

(4) Proxy mechanisms usually require stub generators to generate proxy stubs and marshalling code. These stub generators use interface definitions that are defined outside the language or require language modifications to accommodate them.

(5) It is harder to enforce centralized resource control within the system because proxy mechanisms encourage many independent instances of the virtual machine.

The work by Back *et. al.* [3] and Bernadat *et. al.* [4] focuses on the resource control aspects of competing Java applets on a single virtual machine. Their work is integrated into a JVM implementation while our method of resource control is at an operating system level. For their work they trust the byte code verifier.

8. Conclusions

The security provided by our JVM consists of separate hardware protection domains, controlled access between them, and system resource usage control. An important goal of our work was to maintain transparency with respect to Java programs. Our system does not, however, eliminate covert channels or solve the capability confinement and revocation problem.

The confinement and revocation problem are inherent to the Java language. A reference can be passed from one domain to another and revocation is entirely voluntary. These problems can be solved in a rather straightforward manner, but they do violate the transparency requirement. For example, confinement can be enforced by having the Java Nucleus prohibit the passing of references to cells for which the calling domain is not the owner. This could be further refined by requiring that the cell owner should have permission to call the remote method directly when its data is passed over it by another domain. Alternatively, the owner could mark the cells it is willing to share or maintain exception lists for specific domains. Revocation is nothing more that unmapping the cell at hand.

In the design of our JVM we have been very careful to delay expensive operations until they are needed. An example of this is the on-demand remapping of reference values, since most of the time reference variables are never dereferenced. Another goal was to avoid cross-protection domain switches to the Java Nucleus. The most prominent example of this is pointer assignment which is a tradeoff between memory space and security. By maintaining extra, per protection domain, garbage collector state we perform pointer assignments within the same context, thereby eliminating a large number of cross domain calls due to common pointer assignment operations. The amount of state required can be reduced by having the compiler produce hints about the potential sharing opportunities of a variable.

In our current JVM design, resources are allocated and controlled on a per protection domain basis, as in an operating system. While we think this is an adequate protection model, it might prove to be too coarse grained for some applications and might require techniques as suggested by Back *et. al.* [3].

The current prototype implementation shows that it is feasible to build a JVM with hardware separation whose Java XMI overhead is small. Many more

optimizations, as described in this paper, are possible but have not been implemented yet. Most notable is the lack of instruction sharing which can improve the performance considerably since it eliminates the need for XMIs. When these additional optimizations are factored in, we believe that a hardware assisted JVM compares quite well to JVM's using software fault isolation.

The security of our system depends on the correctness of the shared garbage collector. Traditional JVMs rely on the byte code verifier to ensure heap integrity and a single protection domain garbage collector. Our garbage collector allocates memory over multiple protection domains and cannot depend on the integrity of the heap. Especially the latter requires careful analysis of all the attack scenarios. In our design the garbage collector is very conservative with respect to addresses it is given. Each address is checked against tables kept by the garbage collector itself and the protection domain owning the object to prevent masquerading. The instance state splitting according to the Java visibility rules prevents adversaries from rewriting the contents of a shared object. Security sensitive instance state that is shared, and therefore mutable, is considered a policy error or a programming error.

Separating the security policy from the mechanisms allows the enforcement of many different security policies. Even though we restricted ourself to maintaining transparency with respect to Java programs, stricter policies can be enforced. These will break transparency, but provide higher security. An example of this is the opaque object reference sharing. Rather than passing a reference to shared object state, an opaque reference is passed. This opaque reference can only be used to invoke methods on, the object state is not shared and can therefore not be inspected.

The garbage collector, and consequently runtime relocation, have a number of interesting research questions associated with them that are not yet explored. For example, the Java Nucleus is in a perfect position to make global cache optimization decisions because it has an overall view of the data being shared and the XMIs passed between domains. Assigning a direction to the data being shared would allow fine grained control of the traversal of data. For example, a client can pass a list pointer to a server applet which the server can dereference and traverse but the server can never insert one of its own data structures into the list. This is reminiscent of Shapiro's *diminish-grant* model for which confinement has been proven [34].

The Java Nucleus depends on user accessible low-level operating system functionality that is currently only provided by extensible operating systems (*e.g.,* Paramecium, OSKit [19], L4/LavaOS [28], ExOS [15], and SPIN [6]). Implementing the Java Nucleus on a conventional operating system would be considerably harder since the functionality listed above is intertwined in hard coded abstractions that are not easily adapted.

9. Acknowledgments

We would like to thank Paul Karger and Dan Wallach for many helpful discussions; and Charles Palmer, David Safford, Trent Jaeger, Andy Tanenbaum, Rajesh Bordewaker, Matthias Kaiserswerth, Wietse Venema, Jonathan Shapiro, Peter Gutmann, Larry Koved, and the anonymous referees for helpful comments on the paper.

References

1. Y. Aridor, M. Factor and A. Teperman, "cJVM: a Single System Image of a JVM on a Cluster", *Proc. of the 1999 IEEE International Conference on Parallel Processing (ICPP'99)*, Aizu-Wakamatsu City, Japan, Sep. 1999, 4-11.
2. K. Arnold and J. Gosling, *The Java Programming Language Second Edition*, Addison Wesley, Reading, MA, 1997.
3. G. Back, P. Tullman, L. Stoller, W. C. Hsieh and J. Lepreau, "Java Operating Systems: Design and Implementation", Tech. Rep. UUCS-98-015, Aug. 1998.
4. P. Bernadat, D. Lambright and F. Travostino, "Towards a Resource-safe Java for Service Guarantees in Uncooperative Environments", *Proc. of the 19th IEEE Real-time Systems Symposium (RTSS'98)*, Madrid, Spain, Dec. 1998.
5. B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy, "Lightweight Remote Procedure Call", *Proc. of the 12th Symposium on Operating System Principles, ACM SIGOPS 23*, 5 (Dec. 1989), 102-113.
6. B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker and C. Chambers, "Extensibility, Safety and Performance in the SPIN Operating System", *Proc. of the 15th Symposium on Operating System Principles, ACM SIGOPS 29*, 5 (Dec. 1995), 267-284.
7. B. W. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ, 1981.
8. H. Boehm and M. Weiser, "Garbage Collection in an Uncooperative Environment", *Software—Practice & Experience 18*, 9 (1988), 807-820.
9. Burroughs, *The Descriptor – a Definition of the B5000 Information Processing System*, Burroughs Corporation, Detroit, MI, 1961.
10. J. S. Chase, H. M. Levy, M. J. Feeley and E. D. Lazowska, "Sharing and Protection in a Single-address-space Operating System", *ACM Transactions on Computer Systems 12*, 4 (Nov. 1994), 271-307.
11. D. Dean, E. W. Felten and D. S. Wallach, "Java Security: From HotJava to Netscape and Beyond", *Proc. of the IEEE Security & Privacy Conference*, Oakland, CA, May 1996, 190-200.
12. J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations", *Comm. of the ACM 9*, 3

(Mar. 1966), 143-155.

13. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten and E. F. Steffens, "On-the-fly Garbage Collection: An Excercise in Cooperation", *Comm. of the ACM 21*, 11 (Nov. 1978), 965-975.

14. D. Doligez and G. Gonthier, "Portable Unobtrusive Garbage Collection for Multiprocessor Systems", *Proc. of the 21st Annual ACM SIGPLAN Notices Symposium on Principles of Programming Languages*, Jan. 1994, 70-83.

15. D. R. Engler, M. F. Kaashoek and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management", *Proc. of the 15th Symposium on Operating System Principles, ACM SIGOPS 29*, 5 (Dec. 1995), 251-266.

16. Esmertec, "Jbed Whitepaper: Component Software and Real-Time Computing", White paper, Esmertec, 1998. (available as http://www.jbed.com/).

17. E. Felten, Java's Security History, (available as http://www.cs.princeton.edu/sip/history.html), 1999.

18. B. Ford and J. Lepreau, "Evolving Mach 3.0 to a Migrating Thread Model", *Proc. of the Usenix Winter '94 Conference*, San Francisco, CA, Jan. 1994, 97-114.

19. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin and O. Shivers, "The Flux OSKit: A Substrate for Kernel and Language Research", *Proc. of the 16th Symposium on Operating System Principles, ACM SIGOPS 31*, 5 (Oct. 1997), 38-51.

20. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, MA, 1983.

21. J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison Wesley, Reading, MA, 1996.

22. S. B. Guthery and T. M. Jurgensen, *Smart Card Developer's Kit*, Macmillian Technical Publishing, Indianapolis, IN, 1998.

23. C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu and T. Von Eicken, "Implementing Multiple Protection Domains in Java", *Proc. of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998, 259-270.

24. W. C. Hsieh, M. F. Kaashoek and W. E. Weihl, "The Persistent Relevance of IPC Performance: New techniques for Reducing the IPC Penalty", *Proc. Fourth Workshop on Workstation Operating Systems*, Napa, California, Oct. 1993, 186-190.

25. R. Jones and R. Lins, *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*, John Wiley & sons, New York, 1996.

26. H. T. Kung and S. W. Song, "An efficient parallel garbage collection system and its correctness proof", *IEEE Symp. on Foundations of Computer Science*, 1977, 120-131.

27. K. Li and P. Hudak, "Memory coherence in shared virtual memory systems", *ACM Transactions on Computer Systems 7*, 4 (Nov. 1989), 321-359.

28. J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam and T. Jaeger, "Achieved IPC Performance (Still The Foundation For Extensibility)", *Proc. of the Sixth Workshop on Hot Topics in Operating Systems (HotOS)*, Chatham (Cape Cod), MA, May 1997, 28-31.

29. D. A. Moon, "Genera Retrospective", *Proc. of the International Workshop on Object Orientation in Operating Systems, IEEE CS*, Palo Alto, CA., Oct. 1991, 2-8.

30. G. J. Myers, "Can Software for SDI Ever be Error-free?", *IEEE computer 19*, 10 (1986), 61-67.

31. G. Nelson, *Systems Programming with Modula-3*, Prentice Hall, Englewood Cliffs, NJ, 1991.

32. T. Saulpaugh and C. A. Mirho, *The Inside JavaOS Operating System*, Addison Wesley, Reading, MA, 1999.

33. J. S. Shapiro, D. J. Farber and J. M. Smith, "The Measured Performance of a Fast Local IPC", *Proc. of the Fifth International Workshop on Object Orientation in Operating Systems*, Seattle, WA, Oct. 1996, 89-94.

34. J. S. Shapiro and S. Weber, "Verifying Operating System Security", MS-CIS-97-26, University of Pennsylvania, Philadelphia, PA, July 1997.

35. E. G. Sirer, Security Flaws in Java Implementations, (available as http://kimera.cs.washington.edu/flaws/index.html), 1997.

36. G. L. Steele, "Multiprocessing compactifying garbage collection", *Comm. of the ACM 18*, 9 (Sep. 1975), 495-508.

37. SunSoft, Java Servlet Development Kit, (available as http://java.sun.com/products/servlet/index.html), 1999.

38. Sun Microsystems Inc., *The SPARC Architecture Manual*, Prentice Hall, Englewood Cliffs, NJ, 1992.

39. W. Teitelman, "A tour through Cedar", *IEEE Software 1*, 2 (1984), 44-73.

40. Transvirtual Technologies Inc., Kaffe OpenVM, (available as http://www.transvirtual.com/), 1998.

41. L. Van Doorn, P. Homburg and A. S. Tanenbaum, "Paramecium: An extensible object-based kernel", *Proc. of the Fifth Hot Topics in Operating Systems (HotOS) Workshop*,, Orcas Island, WA, May 1995, 86-89.

42. D. Wetherall and D. L. Tennenhouse, "The ACTIVE IP Option", *Proc. of the Seventh SIGOPS European Workshop, ACM SIGOPS*, Connemara, Ireland, Sep. 1996.

43. N. Wirth and J. Gütknecht, *Project Oberon, The Design of an Operating System and Compiler*, ACM Press, 1992.

Encrypting Virtual Memory

Niels Provos
Center for Information Technology Integration
University of Michigan
provos@citi.umich.edu

Abstract

In modern operating systems, cryptographic file systems can protect confidential data from unauthorized access. However, once an authorized process has accessed data from a cryptographic file system, the data can appear as plaintext in the unprotected virtual memory backing store, even after system shutdown. The solution described in this paper uses swap encryption for processes in possession of confidential data. Volatile encryption keys are chosen randomly, and remain valid only for short time periods. Invalid encryption keys are deleted, effectively erasing all data that was encrypted with them. The swap encryption system has been implemented for the UVM [7] virtual memory system and its performance is acceptable.

1 Introduction

Many computer systems employ cryptographic file systems, *e.g.* CFS [4], TCFS [6] or encryption layers [19], to protect confidential data from prying eyes. A user without the proper cryptographic key is unable to read the contents of the cryptographic file system, nor is he able to glean any useful information from it. However, backing store of the virtual memory system is generally unprotected. Any data read by a process that was originally encrypted can be found as plaintext in swap storage if the process was swapped out. It is possible for passwords and pass phrases to reside in swap long after they have been typed in, even across reboots.

A user expects that all confidential data vanishes with process termination, and is completely unaware that data can remain on backing store. And even if she were aware of it, there is next to nothing she can do to prevent its exposure.

If the integrity of the operating system is compromised and an untrusted party gains root privileges or physical access to the machine itself, she also gains access to the potentially sensitive data retained in backing store.

Our solution to this problem is to encrypt pages that need to be swapped out. These pages are decrypted when they are brought back into physical memory, *e.g.* due to a page fault. After a process terminates, all its pages stored on backing store are invalid, so there is no need to be able to decrypt them; on the contrary, nobody should be able to decrypt them. This suggests the use of volatile random keys that exist only for short time periods.

The remainder of this paper is organized as follows. Section 2 provides further motivation for encrypting the backing store and describes related work. In Section 3 we give a brief overview of virtual memory, note a security problem of secondary storage, and discuss how it can be resolved with encryption. Section 4 explains how we implemented swap encryption. In Section 5 we analyse how the paging times and system throughput are affected. Finally, we conclude in Section 6.

2 Related Work

Computer systems frequently process data that requires protection from unauthorized users. Often it is enough to use access control mechanisms of the operating system to determine who may access specific data. In many cases a system also needs to be secured against physical attacks or protected against security compromises that allow the circumvention of access controls. Blaze addresses data protection with a cryptographic file system called CFS by encrypting all file system data, preventing anyone without the proper cryptographic key from

accessing its content [4]. Anderson, Needham and Shamir aim at hiding the existence of data from an attacker by using a "Steganographic File System" [1]. A cryptographic key and the knowledge that a file exists are needed to access a file's contents. However, security depends on the whole system, and an investigation of the interaction with other system components is essential.

Neither paper looks carefully at its operating environment, nor do they take into consideration that confidential data might inadvertently end up in backing store. The storage of confidential data on a swap device may defeat the purpose of encryption in CFS. Swap data can also be used to reconstruct what files are present in a system, thus defeating the purpose of steganography.

Swap encryption is meant to protect confidential data left on the backing store from intruders who have gained physical access to the storage medium. We observe that the same can be achieved by deleting all confidential data once it is no longer referenced. However, Gutmann has shown that it is difficult to delete thoroughly information from magnetic media or random-access memory [16]. He states: "the easiest way to solve the problem of erasing sensitive information from magnetic media is to ensure that it never gets to the media in the first place. Although not practical for general data, it is often worthwhile to take steps to keep particularly important information such as encryption keys from ever being written to disk."

Schneier and Kelsey describe a secure log system that keeps the contents of the log files confidential even if the system has been compromised [24]. While swap encryption is quite different from secure logging, the attack scenario and operating environment is similar.

There are other systems that modify the paging behavior of a virtual memory system. Notably, Fred Douglis' compression cache compresses memory pages to avoid costly disk accesses [10].

3 Virtual Memory System

One purpose of virtual memory is to increase the size of the address space visible to processes by caching frequently-accessed subsets of the address space in physical memory [2]. Data that does not fit in physical memory is saved on secondary storage known as the backing store. Paged out memory is restored to physical memory when a process needs to access it again [7].

In many operating systems, the virtual memory pager daemon is responsible for reading and writing pages to and from their designated backing store. When a page has been written, it is marked as "clean" and can be evicted from physical memory. The next time a process accesses the virtual memory that was associated with this page, a page fault occurs.

If the page is still resident in physical memory, it is marked as "recently used," and additionally "dirty" if the page fault is caused by a write access. Otherwise, because the page is no longer resident in physical memory, the pager allocates a page of physical memory and retrieves the data from backing store.

3.1 Secondary Storage

Compared to RAM speeds, secondary storage is usually made up from slow media, *e.g.* raw partitions on disk drives. Unlike primary memory, secondary storage is nonvolatile, and the data stored on it is preserved after a system shutdown. Depending on usage patterns, a swap partition can retain data for many months or even years.

Confidential data in a process' address space might be saved on secondary storage and survive there beyond the expectations of a user. She assumes that all confidential data is deleted with the termination of the process. However, the data found by looking at the content of several swap partitions of machines at the Center of Information Technology Integration included: login passwords[1], PGP pass phrases, email messages, cryptographic keys from ssh-agent, shell command histories, URLs, *etc.*

To avoid this, we developed a system that makes data on the backing store impossible for an attacker to read if it was written a certain time prior to the operating system's compromise.

One approach is to avoid swapping completely by not using secondary storage at all. But this is

[1]The author was amazed to find not only his current password, but also older ones that had not been used for months.

not a general solution, and there are many applications and environments that require a virtual address space bigger than the physical memory present in the system.

An application can prevent memory from being swapped out by using the "mlock()" system call to lock the physical pages associated with a virtual address range into memory [16]. There are several disadvantages with this approach. It requires applications to be rewritten to use "mlock()", which might not be possible for legacy applications or difficult if it requires a complicated analysis of which parts of the memory contain confidential data. In addition, "mlock()" reduces the opportunity of the virtual memory system to evict stale pages from physical memory, which can have a severe impact on system performance.

In general, it is not desirable to prevent the system from swapping memory to the disk. Instead, encryption can be used to protect confidential data when it is written to secondary storage by the pager. A user program could install its own encrypting pager [2]. This would lead to greater complexity, require modification of applications and poses difficult decisions about which cryptosystem to use. If a cryptographic file system like CFS [4] were available, the virtual memory pager could be configured to swap to a file that resided on an encrypted file system.

However, in contrast to common use of encryption [20], we require different characteristics for our cryptographic system:

- When a page on backing store is no longer referenced by its owner, the decryption key for that page should be irretrievably lost after a suitable time period (t_R) has passed.
- Only the virtual memory pager should be able to decrypt data read from the backing store.

Clearly, the best protection is achieved with $t_R = 0$. The decryption key, and indirectly the page's content, is irretrievably removed immediately when the page is no longer referenced. This behavior meets the user's expectation that confidential data in a process' address space is deleted with the termination of the process.

However, this is difficult to achieve, and we have to trade off security against performance. Often, a $t_R > 0$ is still acceptable. In the initial implementation, we only guarantee $t_R \leq$ system uptime, but attempt to minimize the average t_R.

This implies the use of volatile encryption keys, valid maximally for the duration of the system's uptime. Such keys are similar to ephemeral keys used to achieve perfect forward secrecy [9]. A volatile key is completely unrelated to all other keys. Knowledge of it does not allow the decryption of old data on secondary storage. Encryption keys are used only by the virtual memory pager and can be generated on demand when they are required, eliminating the need for complicated key management.

On the other hand, swapping to a cryptographic file system does not fulfill either of the two requirements. Key management is an integral part of an encrypting file system [5]. Consequently, permanent nonvolatile encryption keys are present, making it possible to read the data on the swap storage after the system has been shut down. Furthermore, a user with access rights to the swap file on the encrypted file system - usually the root user - can directly read its contents.

Instead, we employ encryption at the pager level. Pages that are swapped out are (optionally) encrypted, and encrypted pages that are read from secondary storage are decrypted.

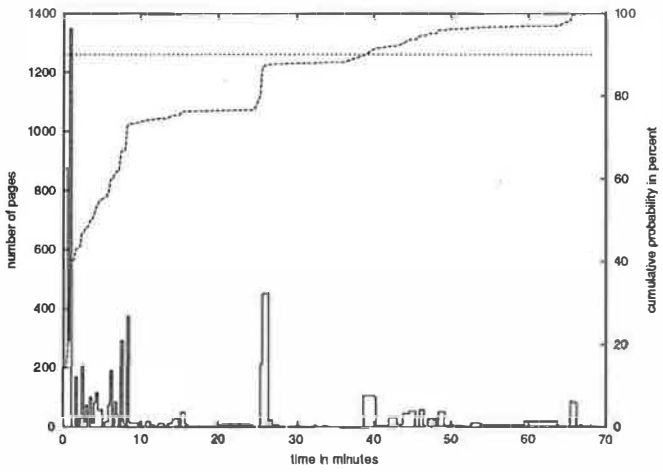


Figure 1: Histogram of page residency in secondary storage for a desktop session and corresponding cumulative probability.

We compared page encryption to zeroing a page on the backing store after it is dereferenced. To get a better understanding of the overheard incurred by such a measure, we recorded how long pages reside

on backing store. Figure 1 shows the result for a desktop session.

Most pages remain in the backing store for only a few minutes. The strong temporal correlation between swapping and zeroing can result in unnecessary cleaning of pages that will be overwritten immediately, and will impact on system performance due to expensive write operations. Zeroing pages also fails to protect against physical attacks that prevent writes to secondary storage, *e.g.* an attacker stealing disks or turning off the system's power supply.

In summary, encryption has the following advantages over physically zeroing pages on the backing store.

- Deleting data by erasing it on disk incurs extra seek time and additional I/O for writing. On the other hand, with encryption the content of a page disappears when its respective encryption key is deleted. Furthermore, encrypting a page is fast compared to writing it, and the encryption cost is spread evenly over the whole swapping process.
- Encryption provides better protection against physical attacks. Mere possession of the disk drive is not sufficient to read its content. The correct encryption key is required, but many physical attacks disrupt the operation of the machine; the content of physical memory is lost, and thus also the encryption key. Additionally, encryption prevents "compromising emanations" caused by data transfers to secondary storage, *i.e.* electromagnetic radiation that carries sensitive information and can be received remotely [11].
- Reliably deleting data from magnetic media is difficult, a problem that does not apply when using encryption [16].

In the next section, we describe our implementation of swap encryption.

4 Swap Encryption

Swap encryption divides naturally into two separate functions: encryption and decryption. The former requires a policy decision about when to encrypt pages. The latter requires knowing which pages read from swap need to be decrypted. The encryption policy can be very simple, *e.g.* all pages that go to swap will be encrypted. A more sophisticated policy might encrypt only pages of processes that have read data from a cryptographic file system. The enumeration of such policies is the subject of future work.

In all cases, though, the decryption is completely independent from the decision to encrypt. For that reason, we keep a bitmap in the swap device that indicates for each page whether it needs to be decrypted after it has been read. Thus, it is possible to change the encryption policy during the runtime of the system without affecting the decryption of pages that have been encrypted while a different policy was in effect.

To achieve lower upper bounds on the window of vulnerability (t_R), we divide the backing store into sections of 512 KByte[2], and give each section its own key. A key consists of a 128-bit encryption key, a reference counter and an expiration time. For a backing store of 256 MByte, keys occupy 14 KByte of memory.

A section's 128-bit cryptographic key is created randomly the first time it is needed, and its reference counter is set to 0. Each time a new page is encrypted with it, the counter is incremented.

When a page is freed on the backing store, the reference counter of the respective key is decremented. A key is immediately deleted when the reference counter reaches 0. Thus, all data encrypted with that key can no longer be decrypted and is effectively erased.

At the moment the first page in a section becomes unreferenced, its encryption key is set to expire after a time period t_R. After t_R has been reached, all pages that reference it have to be re-encrypted with a new key. The number of pages that need to be processed is bounded by the section size, so that the additional encryption overhead is configurable.

The framework for expiration exists, but we have yet to implement re-encryption. However, once this has been done, we can make stricter guarantees for the time that pages remain readable on the backing

[2]The section size is configurable, and depends on how much memory is available for cryptographic keys.

store.

Figure 2 describes the paging process in several steps, and shows where encryption and decryption take place:

1. A user process references memory.
2. If the referenced address has a valid mapping, the data is accessed from the mapped physical page.
3. If the referenced address has an invalid mapping, a page fault occurs.
4. The pager reads the corresponding page from secondary storage.
5. The page is decrypted if its entry in the bitmap indicates that it is encrypted.
6. Finally, the page is mapped into physical memory, and the page fault is resolved.
7. Conversely, if the page daemon decides to evict a page from physical memory,
8. the pager encrypts the page with the encryption key of the section that the page belongs to.
   (a) If the section does not have an encryption key, *e.g.* it is the first encryption, a volatile encryption key is initialized from the kernel's entropy pool.
9. Afterwards, the page is written to secondary storage.

There is one central difference between page encryption and decryption. Pages can be decrypted in place because immediately after they have been read into memory, no process is allowed to access these pages until they have been decrypted. On the other hand, even after a page has been swapped out, a process may access it at any time. This precludes in-place encryption. Instead, we have to allocate pages into which to store temporarily the encryption result, placing additional pressure on the already memory limited VM system.

The volatile keys are stored in an unmanaged part of the kernel memory. As a result, they are never paged out.

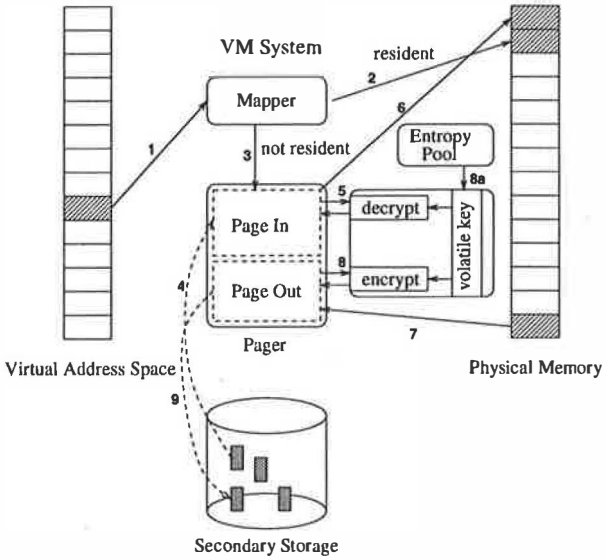


Figure 2: An overview of the swap encryption process.

4.1 Cipher Selection

To be suitable for swap encryption, a cipher needs to fulfill at least three important criteria:

- Encryption and decryption need to be fast compared to disk I/O, so that the encryption does not become the limiting factor in the swapping process.
- The generation of a cipher's key schedule should be inexpensive compared to encrypting a page, so that changing the key schedule does not affect performance. The key schedule of a cipher is usually larger than its encryption key. To conserve system memory we should recompute it every time we switch encryption keys, *e.g.* the encryption key changes when pages are written to different sections.
- The cipher has to support encryption and decryption on a page by page basis, since page in and page out are not sequential. This precludes the use of a stream cipher.

Initially, we planned to employ Schneier's Blowfish encryption algorithm [23]. Its software implementation is very fast, and it has been in use for several years without any apparent security flaws. Nonetheless, Blowfish has one critical drawback. The computation of its key schedule is very expensive, and requires more than 4 KByte of memory.

For that reason, computing the key schedule when it is needed is too expensive, and precomputation is not possible due to large memory requirements.

Based on our environmental constraints, the cipher that matches our needs the best is Rijndael [8]. We describe it in the next section.

4.2 Rijndael

Rijndael is one of the finalists in the advanced encryption standard (AES) competition. It is a variable block and key length cipher. In contrast to many other block ciphers, its round transformation does not have the Feistel structure. Instead, the round transformation is composed of distinct layers: a linear mixing layer, a non-linear layer, and a key addition layer. Rijndael's design tries to achieve resistance against all known attacks while maintaining simplicity [8].

Compared to Blowfish, Rijndael is faster in all aspects, but less studied [12]. We decided to use Rijndael with 128-bit blocks and 128-bit keys. With the optimized C implementation by Gladman [13], the encryption key schedule can be computed in 305 cycles on a Pentium Pro; the decryption key schedule costs 1398 cycles. A block can be encrypted in 374 cycles, and block decryption takes 352 cycles.

However, because all encryption and decryption is done on 4 KByte units, the cost of the key schedule computation is amortized. Therefore, even if we change the key schedule every time, the encryption cost is only 375 cycles on average, and for decryption it is 357 cycles.

Normally, the overall performance of an encryption algorithm is influenced by word conversion to accommodate little and big endian architectures. However, because encryption and decryption happen on the same machine, the word order of the algorithm's output is not relevant, and we do not need to take endianness into consideration.

We use Rijndael in cipher-block chaining (CBC) mode. The CBC mode of operation involves the use of a 128-bit initialization vector. Identical plaintext blocks encrypted under the same key but different IVs, produce different cipher blocks. With $c_0 = IV$, the result of the encryption is defined as

$$c_i = E_K(c_{i-1} \oplus x_i),$$

where the x_i are the plaintext and c_i the ciphertext blocks. The decryption is similar

$$x_i = c_{i-1} \oplus E_K^{-1}(c_i).$$

For swap encryption, the initial 128-bit IV is the 64-bit block number to which the page is written, concatenated with its bitwise complement. This ensures that each page is encrypted uniquely.

Caution is indicated because changing the IV in sequential increments for adjacent pages may result in only small input differences to the encryption function. The attacks described in "From Differential Cryptanalysis to Ciphertext-Only Attacks" [3] might apply in such a situation. For that reason, we encrypt the block number and use that for the IV. Biryukov and Kushilevitz also state, "Another method of IV choice is the encryption of the datagram sequence numbers [...], and sending [the] IV in [the] clear (explicit IV method) [...]. This method is also very vulnerable to our analysis, [...]." Nevertheless, in our case the IV is not explicit, and no IV differences can be observed directly.

4.3 Pseudo-random Generator

To initialize a volatile encryption key we require a source of random bits. The generation of randomness with deterministic computers is very hard. In particular, we do not strive to create perfect randomness characterized by the uniform distribution. Instead, we use pseudo-random generators.

A pseudo-random generator has the goal that its output is computationally indistinguishable from the uniform distribution, while its execution must be feasible [14]. A pseudo-random generator is realized by a stretching function g that maps strings of length n to strings of length $l(n) > n$. If X is a random variable uniformly distributed on strings of length n then $g(X)$ appears to be uniformly distributed on strings of length $l(n)$ [18].

For our purpose, we use the pseudo-random number generator (PRNG) provided by the OpenBSD kernel [21]. The PRNG is a cryptographic stream cipher that uses a source of strong randomness[3] for

[3]The term "source of strong randomness" represents a generator whose output is not really random, but depends on so many entropy providing physical processes that an attacker can not practically predict its output.

initialization and reseeding. This source is referred to as the "entropy pool."

Nonetheless, the problem on how to accumulate strong randomness for the entropy pool remains. Fortunately, a multi-user operating system has many external events from which it can derive some randomness. Gutmann describes a generic framework for a randomness pool [17].

In OpenBSD, the entropy pool

$$P := \{p_1, p_2, \ldots, p_{128}\}$$

consists of 128 32-bit words. To increase the pool's randomness the kernel collects measurements from various physical events: the inter-keypress timing from terminals, the mouse interrupt timing and the reported position of the mouse cursor, the arrival time of network packets, and the finishing time of disk requests.

The measured values from these sources are added to the entropy pool by a mixing function. For each value, the function replaces one word in the pool as follows:

$$p_i \leftarrow u \oplus p_{i+99} \oplus p_{i+59} \oplus p_{i+31} \oplus p_{i+9} \oplus p_{i+7} \oplus p_i,$$

where i is the current position in the pool, and u the 32-bit word that is added. Index addition is modulo 128. After a value has been added i is decremented. To estimate the randomness in the pool, the entropy is measured by a heuristic based on the derivatives of differences in the input values.

A random seed is extracted from the entropy pool as follows: First, the concatenation of $p_1 p_2 \ldots p_{128}$ is given as input to an MD5 hash [22]. Second, the internal state of the MD5 hash for the previous computation is added into the entropy pool. Third, the resulting pool is fed once more into the MD5 hash. Finally, the message digest is calculated. The output is "folded" in half by XOR-ing its upper and lower word. The resulting 64 bits are returned as the seed.

The stretching function is implemented by ARC4, a cipher equivalent to RSADSI's RC4 [25]. The cipher has an internal memory size of $M = n2^n + 2n$, with in our case $n = 8$. We use the random seeds extracted from the entropy pool to initialize the M bits. The output of RC4 is expected to cycle after 2^{M-1} iterations. However, Golić showed that a correlation between the second binary derivative of the least significant bit output sequence and 1 can be detected in significantly fewer iterations [15], which allows the differentiation of *RC4* from a uniform distribution. We can avoid this problem by reseeding RC4's internal state before the number of critical iterations has been reached. In fact, the implementation in OpenBSD reseeds the ARC4 every time enough new entropy has been accumulated.

The kernel provides the "arc4random(3)" function to obtain a 32-bit word from the pseudo-random number generator.

The volatile key of a section is created by filling it with the output from "arc4random(3)." We hope that between the time the system has been booted and the first swap encryption sufficient randomness is available in the kernel entropy pool to ensure good randomness in the RC4 output. Nonetheless, it should be noted that this construction does not create a provably pseudo-random generator as described in the beginning of this section.

5 Performance Evaluation

In the following, we analyse the effect of swap encryption on the paging behavior. We look at page encryption and decryption times, and assess the runtime of applications with large working sets.

All measurements were performed on an OpenBSD 2.6 system with 128 MByte main memory and a 333 MHz Celeron processor. The swap partition was on a 6 GByte Ultra-DMA IDE disk, IBM model DBCA-206480 running at 4200 revolutions per minute. The operating system can sustain an average block write rate of 7.5 MByte/s and a block read rate of 6.3 MByte/s. OpenBSD uses the UVM [7] virtual memory system.

5.1 Micro Benchmark

Our micro benchmark measures the time it takes to encrypt one page. A test program allocates 200 MByte of memory, and fills the memory sequentially with zeros. Afterwards, it reads the allocated memory from the beginning in sequential order. The process is repeated three times.

We use kernel profiling to measure page encryption frequency, and the cumulative time of the encryption function. The kernel function "swap_encrypt()" is called 155336 times with a cumulative running time of 67.96 seconds. One 4 KByte page could be encrypted in 0.44 ms, resulting in an encryption bandwidth of 8.9 MByte/s. The total amount of memory encrypted is 600 MByte.

In UVM, writes to the backing store are asynchronous and reads are synchronous. To determine if I/O is still the bottleneck of the swapping process, we measured the runtime of the test program for different memory sizes, with and without swap encryption. We measure an increase in runtime of about 14% with encryption. To measure asynchronous writes, we modified the test program to write only to memory. The runtime increase of 26% - 36% is due to allocation of new pages that store the encrypted pages until they are written to disk, thus causing the system to swap more often. Figure 3 shows a graph of the results.

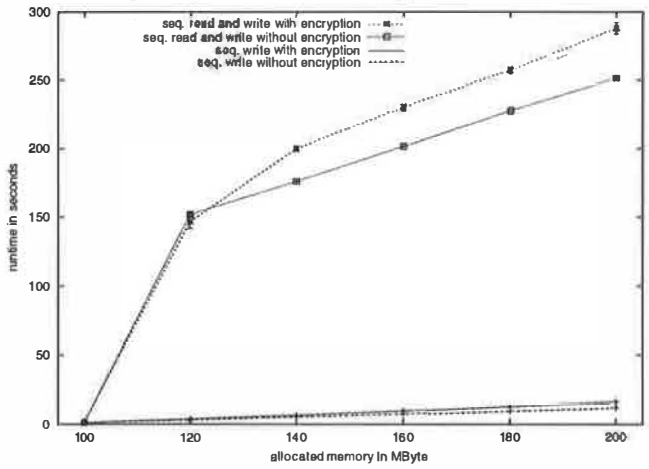


Figure 3: Performance difference between swap encryption and normal swapping when pages are accessed sequentially, illustrating the difference between asynchronous write and synchronous reads.

5.2 Macro Benchmark

To judge the impact of swap encryption on application programs, we used ImageMagick to process a 960 × 1280 image with a 16-bit colorspace. The image was magnified and then rotated by 24°. The runtimes for different magnification factors are shown in Table 1.

	No Encryption		Encryption	
Magnification	Major Faults	Runtime (in sec)	Major Faults	Runtime (in sec)
2.30×	0.4 10^3	49s	0.4 10^3	49s
2.35×	19 10^3	145s	18 10^3	147s
2.40×	22 10^3	169s	22 10^3	180s
2.50×	24 10^3	179s	24 10^3	276s

Table 1: Runtime of image processing tool for different magnification factors.

The table compares the major faults and program runtime for a system that does not use encryption against a system that does. A major fault is a page fault that requires I/O to service it, and does not take into account the pages that have been paged out by the paging daemon.

With increasing magnification factor, the working set size of the program grows larger. We measure a sharp increase of the running time with swap encryption for a magnification factor of 2.5. However, for the other magnification factors the program runtime is not affected that much, even though nearly half of the program's memory was on backing store. Thus, we believe that the overhead caused by encryption is tolerable.

6 Conclusion

Confidential data can remain on backing store long after the process to which the data originally belonged has terminated. This is contrary to a user's expectations that all confidential data is deleted with the termination of the process. An investigation of secondary storage of machines at the Center for Information Technology Integration revealed very confidential information, such as the author's PGP pass phrase.

We investigate several alternative solutions to prevent confidential data from remaining on backing store, *e.g.* erasing data physically from the backing store after pages on it become unreferenced. However, we find that encryption of data on the backing store with volatile random keys has several advantages over other approaches:

- The content of a page disappears when its respective encryption key is deleted, a very fast

operation.

- Encryption provides protection against physical attacks, *e.g.* an attacker stealing the disk that contains the swap partition

Encryption enables us to make the guarantee that unreferenced pages on the backing store become unreadable after a suitable time period upper bounded by system uptime has passed.

We have demonstrated that the performance of our encryption system is acceptable, and it proves to be a viable solution.

The software is freely available as part of the OpenBSD operating sytem and can also be obtained by contacting the author.

7 Acknowledgments

I thank Patrick McDaniel and my advisor Peter Honeyman for careful reviews and helpful comments on the organization of this paper. I also thank Chuck Lever for getting me interested in swap encryption, Artur Grabowski for improving my understanding of UVM and David Wagner for helpful feedback on cipher selection.

References

[1] R. Anderson, R. Needham, and A. Shamir. The Steganographic File System. In *Proceedings of the Information Hiding Workshop*, April 1998.

[2] A. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[3] Alex Biryukov and Eyal Kushilevitz. From Differential Cryptanalysis to Ciphertext-Only Attacks. In *Proceedings of the Advances in Cryptology — CRYPTO '98*, pages 72–88. Springer-Verlag, August 1998.

[4] Matt Blaze. A Cryptographic Filesystem for Unix. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 9–16, November 1993.

[5] Matt Blaze. Key Management in an Encrypting File System. In *Proceedings of the 1994 USENIX Summer Technical Conference*, pages 27–35, June 1994.

[6] G. Cattaneo and G. Persiano. Design and Implementation of a Transparent Cryptographic Filesystem for Unix. Unpublished Technical Report, July 1997. `ftp://edu-gw.dia.unisa.it/pub/tcfs/docs/tcfs.ps.gz`.

[7] Charles D. Cranor and Gurudatta M. Parulkar. The UVM Virtual Memory System. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 117–130, June 1999.

[8] Joaen Daemen and Vincent Rijmen. AES Proposal: Rijndael. AES submission, June 1998. `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/`.

[9] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[10] Fred Douglis. The Compression Cache: Using On-Line Compression to Extend Physial Memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, 1993.

[11] Berke Durak. Hidden Data Transmission by Controlling Electromagnetic Emanations of Computers. Webpage. `http://altern.org/berke/tempest/`.

[12] Niels Ferguson, John Kelsey, Mike Stay, David Wagner, and Bruce Schneier. Improved Cryptanalysis of Rijndael. In *Fast Software Encryption Workshop 2000*, April 2000.

[13] Brian Gladman. AES Algorithm Efficiency. Webpage. `http://www.btinternet.com/~brian.gladman/cryptography_technology/aes/index.html`.

[14] Oded Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudo-randomness*. Springer-Verlag, 1999.

[15] Jovan Dj. Golić. Linear Statistical Weakness of Alleged RC4 Keystream Generator. In *Proceedings of the Advances in Cryptology — Eurocrypt '97*, pages 226–238. Springer-Verlag, May 1997.

[16] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the Sixth USENIX Security Symposium*, pages 77–89, July 1996.

[17] Peter Gutmann. Software Generation of Practially Strong Random Numbers. In *Proceedings of the Seventh USENIX Security Symposium*, pages 243–255, June 1998.

[18] J. Hastad, R. Impagliazzo, L. Levin, and M. Luby. Construction of Pseudorandom Generator from any One-Way Function, 1993.

[19] J. Heidemann and G. Popek. File-System Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[20] Maurice P. Herlihy and J. D. Tygar. How to Make Replicated Data Secure. In *Proceedings of the Advances in Cryptology - CRYPTO '87*, pages 379–391. Springer-Verlag, 1988.

[21] Theo de Raadt, Niklas Hallqvist, Artur Grabowski, Angelos D. Keromytis, and Niels Provos. Cryptography in OpenBSD: An Overview. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 1999.

[22] R. L. Rivest. The MD5 Message Digest Algorithm. RFC 1321, April 1992.

[23] Bruce Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, December 1993.

[24] Bruce Schneier and John Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In *Proceedings of the Seventh USENIX Security Symposium*, pages 53–62, January 1998.

[25] RSA Data Security. The RC4 Encryption Algorithm, March 1992.

Déjà Vu: A User Study Using Images for Authentication*

Rachna Dhamija
rachna@sims.berkeley.edu

Adrian Perrig
perrig@cs.berkeley.edu

SIMS / CS, University of California Berkeley

Abstract

Current secure systems suffer because they neglect the importance of human factors in security. We address a fundamental weakness of knowledge-based authentication schemes, which is the human limitation to remember secure passwords. Our approach to improve the security of these systems relies on *recognition-based*, rather than *recall-based* authentication. We examine the requirements of a recognition-based authentication system and propose Déjà Vu, which authenticates a user through her ability to recognize previously seen images. Déjà Vu is more reliable and easier to use than traditional recall-based schemes, which require the user to precisely recall passwords or PINs. Furthermore, it has the advantage that it prevents users from choosing weak passwords and makes it difficult to write down or share passwords with others.

We develop a prototype of Déjà Vu and conduct a user study that compares it to traditional password and PIN authentication. Our user study shows that 90% of all participants succeeded in the authentication tests using Déjà Vu while only about 70% succeeded using passwords and PINS. Our findings indicate that Déjà Vu has potential applications, especially where text input is hard (e.g., PDAs or ATMs), or in situations where passwords are infrequently used (e.g., web site passwords).
Keywords: Human factors in security, hash visualization, user authentication through image recognition, recognition-based authentication.

*This publication was supported in part by Contract Number 102590-98-C-3513 from the United States Postal Service. The contents of this publication are solely the responsibility of the author and do not necessarily reflect the official views of the United States Postal Service.

1 Introduction

User authentication is a central component of currently deployed security infrastructures. We distinguish three main techniques for user authentication: *Knowledge-based systems*, *token-based systems*, and *systems based on biometrics.*

In today's security systems, knowledge-based schemes are predominantly used for user authentication. Although biometrics can be useful for user identification, one problem with these systems is the difficult tradeoff between impostor pass rate and false alarm rate [DP89]. In addition, many biometric systems require specialized devices, and some can be unpleasant to use.

Most token-based authentication systems also use knowledge-based authentication to prevent impersonation through theft or loss of the token. An example is ATM authentication, which requires a combination of a token (a bank card) and secret knowledge (a PIN).

For these reasons, knowledge-based techniques are currently the most frequently used method for user authentication. In this paper we focus on authentication based on passwords or PINs.

Despite their wide usage, passwords and PINs have a number of shortcomings. Simple or meaningful passwords are easier to remember, but are vulnerable to attack. Passwords that are complex and arbitrary are more secure, but are difficult to remember. Since users can only remember a limited number of passwords, they tend to write them down or will use similar or even identical passwords for different purposes.

One approach to improve user authentication systems is to replace the precise recall of a password or PIN with the recognition of a previously seen image, a skill at

which humans are remarkably proficient. In general, it is much easier to recognize something than to recall the same information from memory without help [Nie93]. Classic cognitive science experiments show that humans have a vast, almost limitless memory for pictures in particular [Hab70, SCH70]. In fact, experiments show that we can remember and recognize hundreds to thousands of pictures in fractions of a second of perception [Int80, PC69]. By replacing precise recall of the password with image recognition, we can minimize the users cognitive load, help the user to make fewer mistakes and provide a more pleasant experience.

The basic concepts of recognition-based authentication are described by Perrig and Song [PS99]. In this paper, however, we explore the user authentication aspects more thoroughly, design the Déjà Vu system, and make the following contributions. First, we perform user studies of a prototype system to validate and improve our image-based user authentication system. Second, we analyze the security of Déjà Vu, discuss possible real-world attacks and illustrate countermeasures.

In the next section we enumerate the shortcomings of password-based authentication. In section 3, we discuss our approach of recognition-based authentication and introduce our solution, Déjà Vu. In section 4, we describe a user study that compares Déjà Vu to traditional authentication methods, and we summarize our findings. Finally, we discuss related work in section 5 and present our conclusions and future work in section 6.

2 Shortcomings of Password-Based Authentication

In this section, we enumerate the problems of password-based authentication, which we address with our work in section 3.

Password and PIN-based user authentication have numerous deficiencies. Unfortunately, many security systems are designed such that security relies entirely on a secret password. Cheswick and Bellovin point out that weak passwords are the most common cause for system break-ins [CB94].

The main weakness of knowledge-based authentication is that it relies on **precise recall** of the secret information. If the user makes a small error in entering the secret, the authentication fails. Unfortunately, precise recall is not a strong point of human cognition. People are much better at imprecise recall, particularly in **recognition** of previously experienced stimuli [Int80, PC69].

The human limitation of precise recall is in direct conflict with the requirements of strong passwords. Many researchers show that people pick easy to guess passwords. For example, an early study by Morris and Thompson on password security found that over 15% of users picked passwords shorter or equal to three characters [MT79]. Furthermore, they found that 85% of all passwords could be trivially broken through a simple exhaustive search to find short passwords and by using a dictionary to find longer ones. They describe an effort to counteract poor passwords, which consists of issuing random pronounceable passwords to users. Unfortunately, the random number generator only had 2^{15} distinct seeds, and hence the resulting space of "random" passwords could be searched quickly. Klein conducted a wide-reaching study of password security in 1989 and notes that 25% of all passwords can be broken with a small dictionary [Kle90].

Other notable efforts to design password crackers were conducted by Feldmeier and Karn [FK89] and Muffett [Muf92]. Because of these password cracker programs, users need to create unpredictable passwords, which are more difficult to memorize. As a result, users often write their passwords down and "hide" them close to their work space. Strict password policies, such as forcing users to change passwords periodically, only increase the number of users who write them down to aid memorability.

As companies try to increase the security of their IT infrastructure, the number of password protected areas is growing. Simultaneously, the number of Internet sites which require a username and password combination is also increasing. To cope with this, users employ similar or identical passwords for different purposes, which reduces the security of the password to that of the weakest link.

Another problem with passwords is that they are easy to write down and to share with others. Some users have no qualms about revealing their passwords to others; they view this as a feature and not as a risk, as we find in the user study discussed in section 4.

The majority of solutions to the problems of weak passwords fall into three main categories. The first types of solutions are proactive security measures that aim to identify weak passwords before they are broken by constantly running a password cracking programs [MT79, FK89]. The second type of solution is also technical in

nature, which utilizes techniques to increase the computational overhead of cracking passwords [Man96]. The third class of solutions involves user training and education to raise security awareness and establishing security guidelines and rules for users to follow [AS99, Bel93].

Note that all three classes of solutions do not remedy the main cause of password insecurity, which is the human limitation of memory for secure passwords. In fact, most previously proposed schemes for knowledge-based user authentication rely on perfect memorization. One exception is the work of Ellsion et al. , which describes a knowledge based authentication mechanism that can tolerate user memory errors [EHMS99]. We discuss these schemes in detail in section 5.

3 Déjà Vu

In this section, we present a solution to address the shortcomings of passwords discussed in the previous section. In particular, we aim to satisfy the following requirements:

- The system should not rely on precise recall. Instead, it should be based on recognition, to make the authentication task more reliable and easier for the user.
- The system should prevent users from choosing weak passwords.
- The system should make it difficult to write passwords down and to share them with others.

3.1 System Architecture

We propose Déjà Vu as a system for user authentication. Déjà Vu is based on the observation that people have an excellent memory for images [Hab70, SCH70, Int80, PC69].

Using Déjà Vu, the user creates an image *portfolio*, by selecting a subset of p images out of a set of sample images. To authenticate the user, the system presents a *challenge set*, consisting of n images. This challenge contains m images out of the portfolio. We call the remaining $n - m$ images *decoy images*. To authenticate, the user must correctly identify the images which are part of her portfolio.

Déjà Vu has three phases: portfolio creation, training, and authentication.

Portfolio Creation Phase

To set up a Déjà Vu image portfolio, the user selects a specific number of images from a larger set of images presented by a server. Figure 2 shows the image selection phase in our prototype.

The type of images used has a strong influence on the security of the system. For example, if the system is based on photographs, it would be easy for users to pick predictable portfolios, to describe their portfolio images and to write down this information and share it with others. For this reason, we use Andrej Bauer's *Random Art* to generate random abstract images [Bau98]. Given an initial seed, *Random Art* generates a random mathematical formula which defines the color value for each pixel on the image plane. The image generation process is deterministic and the image depends only on the initial seed. It is therefore not necessary to store the images pixel-by-pixel, since the image can be computed quickly from the seed. All images are hand-selected to ensure consistent quality.[1]

Figure 1 illustrates sample *Random Art* images and appendix A discusses *Random Art* in more detail. Other methods exist for automatically synthesizing images [Sim91]. We did not explore these and leave this as an area for future study.

Training Phase

After the portfolio selection phase, we use a short *training phase* to improve the memorability of the portfolio images. During training, the user points out the pictures in her portfolio from a challenge set containing decoy images. The selection and the training phase need to occur in a secure environment, such that no other person can see the image portfolio.

[1] From our experience, about 70% of all generated *Random Art* images are aesthetically interesting and are therefore suited for usage in Déjà Vu. The remaining 30% are either too simplistic or fall into a class of images which are frequently generated and visually similar. Since we desire visually distinguishable images in the portfolio and the decoy set, we currently filter out weak images through hand selection.

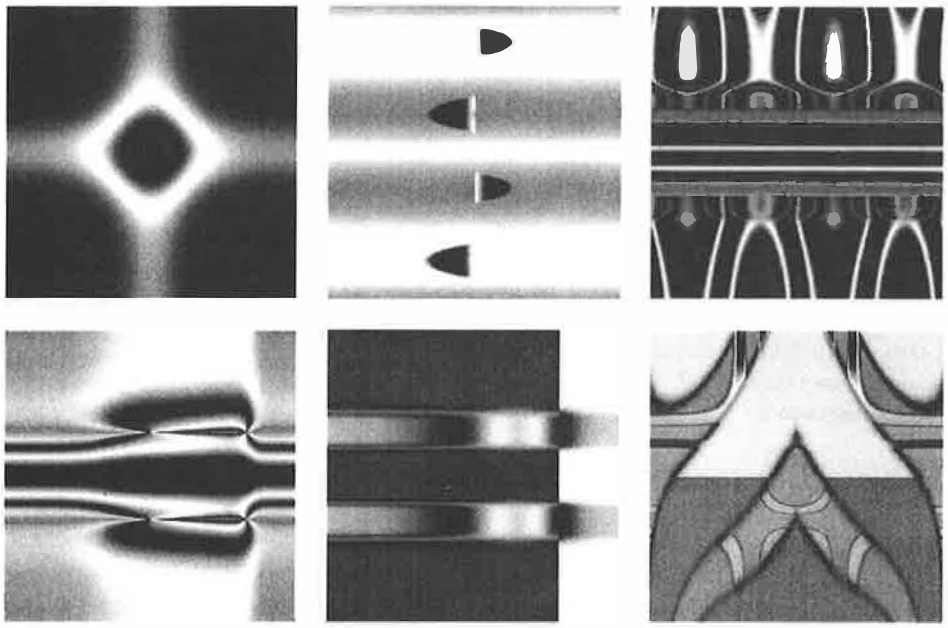


Figure 1: Examples of *Random Art* images

Authentication Phase

A trusted server stores all portfolio images for each user. Since each image is derived directly from the seed, the server only needs to store the seed and not the entire image. In our prototype implementation, the seed is 8 bytes long, hence the storage overhead for each portfolio is small. For each authentication challenge, the server creates a challenge set, which consists of portfolio and decoy images. If the user correctly identifies all portfolio images, she is authenticated.

In general, a weakness of this system is that the server needs to store the seeds of the portfolio images of each user in cleartext. Tricks similar to the hashed passwords in the `/etc/passwd` file do not work in this case, because the server needs to present the portfolio to the user, hidden within the decoy images. For this reason, we assume the server to be secure and trusted, similar to Kerberos [SNS88]. To reduce the trust required from each server, the portfolio can be to split among multiple servers, and each server can contribute a part of the challenge set for each authentication.

3.2 Attacks and Countermeasures

We identify a number of possible attacks which serve to impersonate the user. In the following scenarios, Mallory is an attacker who wants to impersonate Alice.

Brute-force attack. Mallory attempts to impersonate Alice by picking random images in the challenge set, hoping that they are part of Alice's portfolio. The probability that Mallory succeeds is $1/\binom{n}{m}$, which depends on the choice of n, the number of images in the challenge set, and m, the number of portfolio images shown. For example, for $n = 20$ and $m = 5$, we get $1/\binom{20}{5} = 1/15504$, which is equivalent to a four-digit PIN. To prevent brute-force attacks, the system may deny access after a small number of trials.

Educated Guess Attack. If Mallory knows Alice's taste in images he might be able predict which images are in Alice's portfolio.

Our first countermeasure is to use *Random Art*, which makes it hard for Mallory to predict Alice's portfolio images, even if he knows her preferences. Our user study shows that if photographs are used instead of Random Art, it is easier to predict some portfolio images chosen by Alice, given some knowledge about her.

Since users tend to pick the most aesthetically appealing pictures for their portfolios, it will be clear which images in the challenge set are the portfolio images if they are not all equally appealing. We therefore hand select images to ensure that no weak images are used. (We call images *weak*, if no user would select them for their portfolio). Hand selecting images is not a drawback, since a Déjà Vu system can function with a fixed set of images, on the order of 10,000 images.

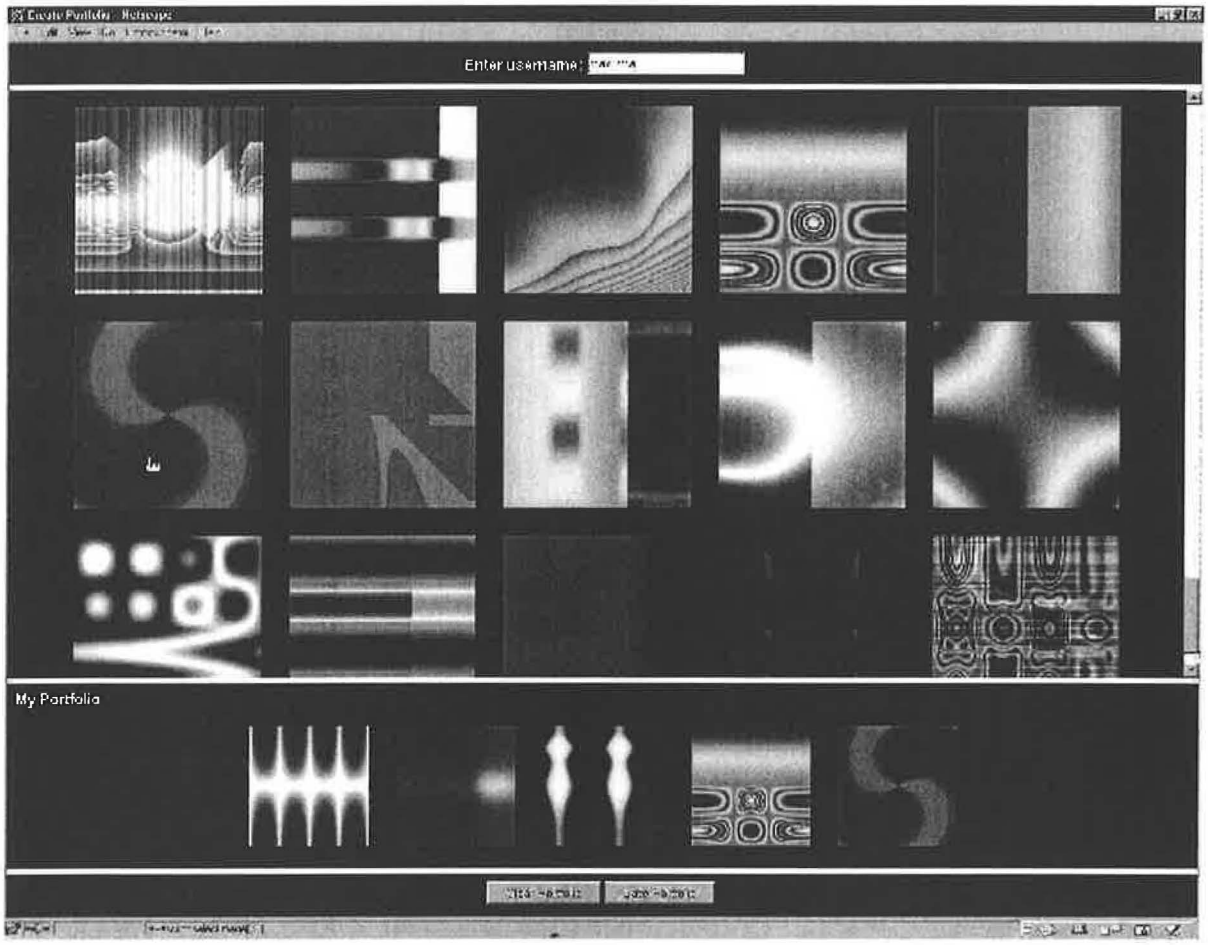


Figure 2: Portfolio selection window

Observer Attacks. Ross Anderson shows that observation of PIN codes on ATMs has been used to impersonate users [And94]. Similarly, if Mallory observes Alice during multiple authentications, he can know Alice's portfolio perfectly. We propose the following countermeasures.

- If the size of Alice's portfolio p is larger than the number of portfolio images in a challenge set m, the probability that an observer sees the same portfolio images after one observation is $1/\binom{p}{m}$. Although the security is still weakened after an observer learns images in a portfolio, an observer still can not impersonate Alice easily.

  Assuming that the images are displayed in a way that only Alice can see them clearly, the observer gains no knowledge of the portfolio by observing which images she selects, since the position of the portfolio images within the challenge set is randomized.

- The method for the image selection is hidden, such that an observer cannot see whether a given image is in the portfolio or not. If the observer cannot see which keys are pressed or can not determine which images are selected, he gets no useful information.

- The portfolio images can be slightly changed in each authentication. The goal is that a legitimate user can still recognize her portfolio images, while leaking less information about the portfolio to an observer. Further study is needed to explore image distortion methods and to determine how modifications in images are perceived by users.

Intersection Attack. If all the portfolio images are part of the challenge set, and all decoy images are changed in

each challenge, Mallory can use the intersection of two challenge sets to reveal the portfolio. This is a serious problem, but we can design a system which can resist this attack through the following countermeasures.

- The same challenge set (portfolio images and decoy images) is always presented to the user. If it remains the same, an intersection attack does not reveal any useful information. The drawback, however, is that since the decoy images remain the same across many login sessions, Alice might start to remember decoy images and flag them as portfolio images in future authentication sessions. Future study is needed to see if this is the case.

- A small number of decoy images remain in the challenge set over multiple authentications. Again, the problem with this approach is that users may learn a decoy image if it is repeated enough times and then mistake it for a portfolio image.

- The authentication can be split up into multiple stages. Each stage presents a challenge set with a random number of portfolio images. If a user makes a mistake in any stage, all subsequent stages will only display decoy images without any portfolio images. This prevents an adversary from performing repeated impersonation attacks to discover the entire portfolio.

- We find in the user study that the failure rate is much lower for Déjà Vu than for password or PIN-based systems. This increased accuracy allows us to tighten the bound on unsuccessful logins before the account is blocked. This, however, opens the door to denial-of-service attacks which may render this method impractical.

Another possibility is to combine the countermeasures such that Mallory does not receive any useful information from multiple unsuccessful logins. First, the system uses the multi-stage authentication, which reveals only decoy images after the user makes an error in any stage. In addition, the system discards portfolio and decoy images that are shown in any unsuccessful login attempt. A shortcoming is that too few images may remain in the portfolio, and the system would need to perform a portfolio replenishment phase after a successful login. Since this takes time and may annoy the user, this method might be impractical. To prevent a denial-of-service attack from depleting the portfolio, the system can disable logins after a small number of unsuccessful login attempts. In case a user successfully authenticates after an unsuccessful attempt, the system can then replace the previously discarded portfolio images and perform a training phase with the images the user forgot.

3.3 Sample Applications

We describe two applications for which Déjà Vu is well suited and would improve security.

Customer Authentication at ATM

Banks face a multitude of problems concerning customer authentication at ATM's. First, many people have problems memorizing their PIN and pick either trivial PINs or write them on the ATM card. Anderson enumerates the many security problems with ATM's [And94].

The main problem for using Déjà Vu for ATM's is the portfolio creation. This is not a problem when customers pick up their card at the bank, since the portfolio selection and training can be done in a secure environment at the bank. If the client receives the ATM card in the mail, the portfolio creation is a more difficult problem. Sending all the images of the portfolio in the mail is not satisfactory, because we want to prevent people from possessing a paper copy of their secret information. Instead, we could use a one-time PIN to bootstrap the system, which the user can authenticate with initially at the ATM, which will then perform the portfolio creation and training.

The seeds of the portfolio images would be stored on a secure server. The authentication process would work as we describe previously. To achieve the same order of security as a four-digit PIN, we can use five images per portfolio and fifteen images in the decoy set. The probability of guessing the correct portfolio is $1/\binom{20}{5} = 1/15504$, which is lower than the $1/10000$ for four-digit PINs.

Web Authentication

The main problem with user authentication on the Web is that many sites are used infrequently and people forget their passwords over time. Another problem is that the number of sites which require a username and password combination to access it are increasing dramatically. The result is that users choose trivial passwords or they pick the same password that they already use for

higher-security applications. Even so, users often forget their passwords; that's why many sites have forgotten-password recovery systems in place.

Déjà Vu is well suited for this problem because the "forgotten-password" recovery rate is very high for *Random Art* images, as we show in the user study. The creation of image portfolios is also easy to accomplish over the web.

4 User Study

We conducted a user study to compare a prototype image authentication system to traditional recall-based authentication systems (passwords and PINs). We compare two types of image portfolios, one using *Random Art* images and another which uses photographs. The user study consists of three phases: interviews, low-fidelity testing and formal prototype testing. In all phases participants were selected to be representative of the general population of computer users. An equal number of novice and expert users were selected, all of who were familiar with password authentication.

4.1 Task Analysis

In order to analyze the task of password authentication, we interviewed thirty people about their password behavior. While the sample size is small, our findings mirror the results of other larger surveys on the subject [AS99].

- We find that while participants have 10 - 50 instances where passwords are required, our users have only 1-7 unique passwords, which they use for multiple situations. Many of these unique passwords are variations on each other to aid memorability.
- Users have a variety of ways for coming up with passwords that they can remember. In most cases, people choose something that is personally meaningful to them (e.g., their own names, family members names, phone numbers, favorite movies). When asked to change passwords, most use a variation on a previous password. The average password length is 6 characters and the majority of passwords are composed of alphabetic characters appended by one or two numerical characters.
- The vast majority of participants write their passwords down (independently of whether they are novices or experts, or have been trained in password security). Some have a policy of writing all passwords down, while others just write down passwords initially until they remember them or only write down infrequently used passwords. Some users store their passwords in PDAs.
- System restrictions do impact password behavior. In general, users expend the minimum effort that is required to manage their passwords. For example, some will only make passwords alphanumeric or insert special characters if required, and most users did not ever change their passwords unless required to do so. However, restrictions do not prevent users from finding workarounds or engaging in other insecure behavior. One user likes having only one password to remember, so when she is required to change any password, she will change all of her other passwords to be the same.
- The level of security education or training also does not appear to have any impact on behavior. Although most users have received some sort of password security training, they ignored it stating that it was too cumbersome or simply not practical to follow.
- Some users who spoke foreign languages reported that they used their own names or words common in their native language as passwords, because "if it is not in English, it is hard for hackers to break". Apparently our users were not aware of the existence of multi-lingual password cracking dictionaries.
- An interesting finding is that people viewed the ability to share passwords with others as a feature. Almost all participants shared their bank PIN with family or friends and several users shared account passwords with others because this was a convenient way to collaborate, share information or transfer files.
- All participants expressed strong feelings of dislike and frustration with their experiences remembering, using and losing passwords. Yet surprisingly, most people preferred them to alternatives. For example many disliked hardware tokens because of experiences losing or misplacing them. A couple of participants who had experience with biometrics (fingerprint readers) felt that these systems were unreliable and performed poorly compared to passwords. Others disliked biometrics because of perceived privacy threats.

4.2 Informal Low-fi Prototype testing

Unlike high fidelity prototypes which are very detailed and may look very much like the final interface, low fidelity or "low-fi" prototypes are a rough rendition of the interface that presents only the main features. Low-fi prototypes are especially useful in early stage interface design to quickly iterate, test and experiment with new designs.

We tested the low-fi prototype to get early feedback on interfaces for portfolio selection and authentication (we did not compare it to text-based authentication at this stage). The low-fi testing also helped us to determine which variations in the *Random Art* algorithm produces the most memorable and distinguishable images and served as a way to preselect the images that would be used for future testing.

4.3 Formal User Testing

We developed a web-based prototype of Déjà Vu that allows users to create image portfolios and to authenticate themselves to the system later by selecting their portfolios from a challenge set. We designed a user study to compare Déjà Vu to standard web authentication using password/PIN dialogues.

We selected twenty participants (11 males and 9 females) to be representative of the general population of computer users. An equal number of novice and expert users were selected, all of who were familiar with password authentication.

The testing consisted of two sessions. During the first session, participants had to create a four digit PIN and a password with a minimum of six characters, both which they believed to be secure and that they had never used before. Other than character length, we imposed no limitations on the type of password or PIN created.

Participants also created two types of image portfolios, one consisting of five *Random Art* images and another consisting of five photographs. We presented each user with the same set of one hundred images to choose from, although the image order was randomized, to see if there was any similarity in the images chosen by users.

From user to user, we varied the order in which passwords, PINs, *Random Art* portfolios and photo portfolios were created to ensure that there was no bias due to task sequence.

Participants next had to authenticate using all four techniques, in the same order that they had created them. This ensured that several minutes and tasks elapsed between each PIN, password and portfolio creation and the login using that technique. To authenticate using image portfolios, users had to select their five portfolio images, which were randomly interspersed with twenty decoy images that were never seen before. (Selecting 5 images form a challenge set of 25 images results in 53,130 possible combinations, which is equivalent to a 4-5 digit PIN.) We gave participants an unlimited amount of time and attempts to login.

The second session occurred one week later and participants once again had to login using all four techniques (i.e., with the PIN, password and portfolios created in the first session). Again, we allowed an unlimited amount of time and number of attempts.

4.4 Task Completion Time and Error Rate

It took longer for users to create image portfolios than to create passwords and PINS. Photo portfolios took longer to create than *Random Art* portfolios, because people spent more time browsing and looking at each image.

Users also required more time to login with image portfolios compared to passwords and PINs. It took slightly longer for users to login using *Random Art* compared to photos, suggesting that people can recognize photographic images more quickly than abstract images.

After one week, however, there was a greater degradation in performance with PINs and passwords compared to portfolios. Table 1 shows the average creation and login times. The reason for the longer than expected login times for passwords and PINs is that several users required multiple attempts. (Note that login times include multiple attempts, but do not include those who could not login at all).

A number of minor and major errors were made with PINs, passwords and portfolios. During the first session all users were able to recover from their errors and to login successfully with portfolios, but this was not always the case with PINs and passwords, no matter how long or how many login attempts were made.

Even after one week, the number of unrecoverable errors made with images was far lower than that of passwords

	PIN	Password	Art	Photo
Create	15	25	45	60
Login	15	18	32	27
Login (after one week)	27	24	36	31

Table 1: Average seconds to create/login

and PINs. If we imposed more secure password and PIN restrictions (e.g., restrictions on character length and type, limited number of attempts), we suspect that the number of failed logins with passwords and PINs would increase. In contrast, all users were able to remember at least four out of five of their portfolio images on the first attempt.

Further study is needed to discover how frequency of use and long term memory effects will influence performance and error rates in portfolio authentication.

4.5 Qualitative Results

Déjà Vu is easier than it looks: Although some users remarked that they would never be able to remember the portfolios they created, all were surprised that they could recognize their images and at how quickly the selection took place. It is interesting to note that after the first week, more users forgot their usernames than their portfolios.

Text vs. images: The majority of users reported that photo portfolios were easier to remember than PINs and passwords, especially after 1 week, and that they would use such a system if they were confident that it was secure and if image selection times were improved.

Random Art vs. photos: Users varied in whether they thought photo or *Random Art* portfolios were easier to use.

Users tend to select photographic images based on a theme or something that has personal meaning to them. (e.g., hobbies, places they have visited). There was much more variation in the *Random Art* images selected by users compared to the photographs. For example, although participants were presented with a choice of 100 images, 9 out of the 20 participants included a photograph of the Golden Gate bridge in their portfolios. In contrast, there were few *Random Art* images that were chosen by more than one user.

After the user testing was complete, users described the portfolios they had chosen. The descriptions of a photograph chosen by more than one user were virtually identical from user to user. However, no two descriptions of a *Random Art* image were alike. Participants found it hard to describe or to recall the *Random Art* images in concrete terms and instead related them to objects or actions (e.g., "it looks like a woman dancing"). For this reason, we conjecture that it would be hard for a third party to identify another's portfolio images based on descriptions or recalled drawings alone. Further study is needed to see if this is the case.

4.6 Interface Issues

Faster image portfolio creation and login will help to make such a system usable. One improvement would be to reduce image size to minimize the need for scrolling, which occupied a significant portion of the task completion time.

Users wished to have more feedback in many instances, but it will be important to give users feedback without compromising security. For example, during portfolio creation and authentication, some users were confused about how many images they had picked thus far if a portfolio window was not available. If portfolios are created in a secure environment, it would be possible to show thumbnails of the selected images. In the case of an insecure environment, simply providing the number of images picked thus far would be an improvement.

5 Related Work

We review previous work which makes an attempt to solve the problem of password-based user authentication.

Blonder patented a "graphical password", which requires a user to touch predetermined areas of an image (tap regions) in a predetermined sequence for authentication [Blo96]. The main drawback to this system is that

	PIN	Password	Art	Photo
Failed Logins	5% (1)	5% (1)	0	0
Failed Logins (after one week)	35% (7)	30% (6)	10% (2)	5% (1)

Table 2: % Failed logins (# failed logins/20 participants)

it is location and sequence dependent, so the user is required to recall the regions to tap and the correct order in which to tap them.

Jermyn, et al. propose a graphical password selection and input scheme, where the password consists of a simple picture drawn on a grid. [JMM+99]. A benefit of their solution is that it removes the need for temporal recall, by decoupling the position of inputs from the temporal order in which those inputs occur. Early cognition experiments do indeed support the claim that pictures are recalled better than words. Their solution, however, still suffers from the fact that it requires users to precisely recall how to draw their images, rather than relying on recognition.

Passlogix Inc. distributes v-go, an application which remembers user names and passwords and automatically logs the user on to password-protected Web sites and applications [Pas00]. They allow users to create passwords by clicking on objects in a graphical window, such as by entering the time on a clock, drawing cards from a card deck, selecting ingredients to mix a cocktail or to cook a meal, dialing a phone number, hiding objects in a room, trading stocks, and entering a password on a keyboard. The weaknesses of their system are manyfold.

First, the space of different passwords is very small. For example, there are only limited places available to select to cook a meal. In the case of hiding objects in a room, the requirement to hide the objects already strongly reduces the state space. It would be better if the user could place objects in arbitrary locations. There are only a few places in the given room where the objects can really be hidden, for example under the mattress or the cabinet are locations which users are likely to select.

Furthermore, the system allows users to pick poor passwords. For example, choosing all aces in a deck of cards is certainly not secure. It is likely that many users will choose commonly known combinations, for example by choosing to mix the same drinks.

Finally, the system requires users to precisely recall the authentication task, instead of relying on recognition. Another weakness is that an attacker will only need to break the v-go password to get access to all the users' other passwords.

IDArts distributes Passfaces, an authentication system based on recognizing previously seen images of faces [Art99]. This idea is similar to ours, and there is strong evidence to support their claim that humans have an innate ability to remember faces. They claim that authentication rates can be significantly improved by "training" the user during passface creation, which we did not do in our study.

A drawback of their system is that users pick faces which they are attracted to, which greatly facilitates impersonation attacks. Interestingly, in our study many users told us that they did not select photographs of people because they did not feel that they could relate personally to the image. We did notice that when pictures of people were chosen, the people closely resembled the users (e.g., one user selected an image that resembled his grandparents, one Indian woman selected an image of an Indian woman and a Chinese woman selected an image of a Chinese man). Since we use randomly generated images, knowing the preferences of a person only has limited usefulness.

Ellison et al. propose a scheme in which a user can protect a secret key using "the personal entropy in his whole life", that is by encrypting the passphrase using the answers to several personal questions [EHMS99]. The scheme is designed so that a user can forget the answers to a subset of the questions and still recover the secret key, while an attacker must learn the answer to a large subset of the questions to learn the key.

Naor and Shamir propose a Visual Cryptography scheme, which splits secret information into two transparencies, such that each part contains no useful information, but the combination reveals the secret [NS95]. Naor and Pinkas extend this idea as a means for a user to authenticate text and images [NP97]. In this case, the recipient is equipped with a transparency. When the recipient places the transparency over a message or image that was sent to him, the combination of both images reveals the message. Visual cryptography could be used to devise a user authentication scheme that is token based.

Ian Goldberg's "visual key fingerprint"[Gol96] and Raph Levien's [Lev96] PGP Snowflake were developed as a way to graphically identify and recognize PGP key fingerprints.

Adams and Sasse propose that educating users in security is a solution for the problem of choosing weak passwords [AS99]. They claim that if users receive specific security training and understand security models, they will select secure passwords and refrain from engaging in insecure behavior. In our user study, however, we discover that the level of security training did not prevent users from choosing trivial passwords or from storing them insecurely. We conjecture that this is the case because people prefer convenience over security. Therefore, security should be an inherent component of the system by default.

6 Conclusions and Future Work

Previous research recognized the weaknesses of knowledge-based authentication schemes (in particular password-based computer logins). So far, however, most of the proposed solutions have been based on technical fixes or on educating users. Neither of these address the fundamental problem of knowledge-based authentication systems, which is that the authentication task is based on precise recall of the secret knowledge.

Since people are much better at recognizing previously seen images than at precisely recalling pass phrases from memory, we employ a recognition-based approach for authentication. We examine the requirements of a recognition-based system and propose Déjà Vu, in which we replace the precise recall of pass phrases with the recognition of previously seen images. This system has the advantage that the authentication task is more reliable, easier and fun to use. In addition, the system prevents users from choosing weak passwords and makes it difficult for users to write passwords down and to communicate them to others.

We conducted a user study which compares Déjà Vu to traditional password and PIN authentication. Results indicate that image authentication systems have potential applications, especially where text input is hard (e.g., PDAs or ATMs), for infrequently used passwords or in situations where passwords must be frequently changed. Since the error recovery rate was significantly higher for images, compared to passwords and PINS, such a system may be useful in environments where high availability of a password is paramount and where the difficulty to communicate passwords to others is desired. Further study is required to determine how user performance and error rate will vary with frequency of use, over longer time periods and with large or multiple portfolios.

Many improvements can be made to strengthen the system against attack and to improve its usability. For example, we are exploring ways to mask or distort portfolio images, such that users will be able to recognize their images, while leaking information about the portfolio to observers. We are also exploring authentication schemes that take advantage of other innate human abilities (e.g., spatial navigation).

Hackers recognize that humans are often the weakest link in system security and exploit this using social engineering tactics[Kni94]. Yet designers do not always include human limitations in their evaluation of system security. Systems should not only be evaluated theoretically, but by how secure they are in common practice.

Acknowledgments

We would like to thank Doug Tygar, James Landay, and John Canny for their encouragement and advice. We would also like to thank Dawn Song and Ben Gross for their valuable feedback. Furthermore, we would like to thank the anonymous reviewers for their valuable comments and suggestions.

References

[And94] Ross J. Anderson. Why Cryptosystems Fail. *Communications of the ACM*, 37(11):32–40, November 1994.

[Art99] ID Arts. `http://www.id-arts.com/technology/papers/`, 1999.

[AS99] Anne Adams and Martina Angela Sasse. Users are not the enemy: Why users compromise computer security mechanisms and how to take remedial measures. *Communications of the ACM*, 42(12):40–46, December 1999.

[Bau98] Andrej Bauer. Gallery of random art. WWW at `http://andrej.com/art/`, 1998.

[Bel93] W. Belgers. Unix password security, 1993.

[Blo96] G. Blonder. United states patent, 1996. United States Patent 5559961.

[CB94] B. Cheswick and S. Bellovin. Firewalls and internet security: Repelling the wily hacker, 1994.

[Dha00] Rachna Dhamija. Hash visualization in user authentication. In *Proceedings of the Computer Human Interaction 2000 Conference*, April 2000.

[DP89] D. W. Davies and W. L. Price. *Security for Computer Networks*. John Wiley and Sons, 1989.

[EHMS99] Carl Ellison, Chris Hall, Randy Milbert, and Bruce Schneier. Protecting secret keys with personal entropy. to appear in Future Generation Computer Systems, 1999.

[FK89] D. C. Feldmeier and P. R. Karn. UNIX password security—ten years later (invited), 1989. Lecture Notes in Computer Science Volume 435.

[Gol96] Ian Goldberg. Visual key fingerprint code. Available at `http://www.cs.berkeley.edu/iang/visprint.c`, 1996.

[Hab70] Ralph Norman Haber. How we remember what we see. *Scientific American*, 222(5):104–112, May 1970.

[Int80] Helene Intraub. Presentation rate and the representation of briefly glimpsed pictures in memory. *Journal of Experimental Psychology: Human Learning and Memory*, 6(1):1–12, 1980.

[JMM+99] Ian Jermyn, Alain Mayer, Fabian Monrose, Michael K. Reiter, and Aviel D. Rubin. The design and analysis of graphical passwords. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[Kle90] Daniel Klein. A survey of, and improvements to, password security. In *Proceedings of the USENIX Second Security Workshop, Portland, Oregon*, 1990.

[Kni94] The Knightmare. *Secrets of a Super Hacker*. Loompanics Unlimited, Port Townsend, Washington, 1994.

[Lev96] Raph Levien. Pgp snowflake. Personal communication, 1996.

[Man96] Udi Manber. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers and Security*, 15(2):171–176, 1996.

[MT79] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11), Nov 1979.

[Muf92] D. Muffett. Crack: A sensible password checker for unix, 1992. A document distributed with the Crack 4.1 software package.

[Nie93] Jakob Nielsen. *Usability Engineering*. Academic Press, 1993.

[NP97] M. Naor and B. Pinkas. Visual authentication and identification. In Burt Kaliski, editor, *Advances in Cryptology - Crypto '97*, pages 322–336, Berlin, 1997. Springer-Verlag. Lecture Notes in Computer Science Volume 1294.

[NS95] M. Naor and A. Shamir. Visual cryptography. In Alfredo De Santis, editor, *Advances in Cryptology - EuroCrypt '94*, pages 1–12, Berlin, 1995. Springer-Verlag. Lecture Notes in Computer Science Volume 950.

[Pas00] Passlogix. v-go. WWW at `http://www.passlogix.com/`, 2000.

[PC69] A. Paivio and K. Csapo. Concrete image and verbal memory codes. *Journal of Experimental Psychology*, 80(2):279–285, 1969.

[PS99] Adrian Perrig and Dawn Song. Hash visualization: A new technique to improve real-world security. In *Proceedings of the 1999 International Workshop on Cryptographic Techniques and E-Commerce (CryTEC '99)*, 1999.

[SCH70] L. Standing, J. Conezio, and R.N. Haber. Perception and memory for pictures: Single-trial learning of 2500 visual stimuli. *Psychonomic Science*, 19(2):73–74, 1970.

[Sim91] Karl Sims. Artificial evolution for computer graphics. In Thomas W. Sederberg, editor, *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics (SIGGRAPH '91)*, pages 319–328, Las Vegas, Nevada, USA, July 1991. ACM Press.

[SNS88] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Conference Proceedings*, pages 191–200, 1988.

[WT99] Alma Whitten and J. D. Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

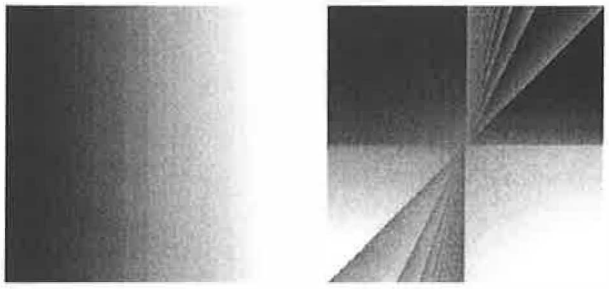
A *Random Art*

One proposed hash visualization algorithm is Random Art, a technique that converts meaningless strings into abstract structured images. *Random Art* was developed by Andrej Bauer, and is based on an idea of genetic art by Michael Witbrock and John Mount. Originally *Random Art* was conceived for automatic generation of artistic images. A brief overview and demonstration of *Random Art* can be found at Andrej's *Random Art* web site [Bau98].

The basic idea is to use a binary string s as a seed for a random number generator. The randomness is used to construct a random expression which describes a function generating the image—mapping each image pixel to a color value. The pixel coordinates range continuously from -1 to 1, in both x and y dimensions. The image resolution defines the sampling rate of the continuous image. For example, to generate a 100×100 image, we sample the function at 10000 locations.

Random Art is an algorithm such that given a bit-string as input, it will generate a function $\mathcal{F} : [-1,1]^2 \to [-1,1]^3$, which defines an image. The bit-string input is used as a seed for the pseudo-random number generator, and the function is constructed by choosing rules from a grammar depending on the value of the pseudo-random number generator. The function $\mathcal{F}$ maps each pixel (x, y) to a RGB value (r,g,b) which is a triple of intensities for the red, green and blue values, respectively. For example, the expression $\mathcal{F}(x, y) = (x, x, x)$ produces a horizontal gray grade, as shown in figure 3(a). A more complicated example is the following expression, which is shown in figure 3(b).

$$\text{if } xy > 0 \text{ then } (x, y, 1) \text{ else } (\text{fmod}(x, y), \text{fmod}(x, y), \text{fmod}(x, y)) \quad \text{(A.1)}$$



(a) Image for expression (x, x, x)

(b) Image for expression (A.1)

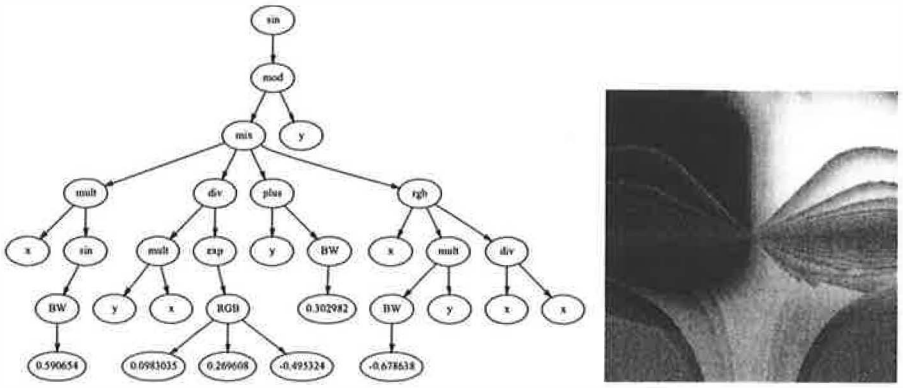
Figure 3: Examples of images and corresponding expressions.

The function $\mathcal{F}$ can also be seen as an expression tree, which is generated using a *grammar* G and a *depth parameter* d, which specifies the minimum depth of the expression tree that is generated. The grammar G defines the structure of the expression trees. It is a version of a context-free grammar, in which alternatives are labeled with probabilities. In addition, it is assumed that if the first alternative in the rule is followed repeatedly, a terminal clause is reached. This condition is needed when the algorithm needs to terminate the generation of a branch. For illustration, consider the following simple grammar:

$$E ::= (C, C, C)^{(1)}$$
$$A ::= \langle \text{random number} \in [-1,1] \rangle^{(\frac{1}{3})} \mid x^{(\frac{1}{3})} \mid y^{(\frac{1}{3})}$$
$$C ::= A^{(\frac{1}{4})} \mid \text{add}(C, C)^{(\frac{3}{8})} \mid \text{mult}(C, C)^{(\frac{3}{8})}$$

The numbers in subscripts are the probabilities with which alternatives are chosen by the algorithm. There are three rules in this simple grammar. The rule E specifies that an expression is a triple of compound expression C. The rule C says that every compound expression C is an atomic expression A with probability $\frac{1}{4}$, or either the function add or mult applied to two compound expressions, with probabilities $\frac{3}{8}$ for each function. An atomic expression A is either a constant, which is generated as a pseudorandom floating point number, or one of the coordinates x or y. All functions appearing in the *Random Art* algorithm are scaled so that they map the interval $[-1,1]$ to the interval $[-1,1]$. This condition ensures that all randomly generated expression trees are valid. For example, the scaling for the add function is achieved by defining $\text{add}(x, y) = (x + y)/2$.

The grammar used in the *Random Art* implementation



(a) *Random Art* expression tree

(b) Generated image

Figure 4: *Random Art* expression tree and the corresponding image

is too large to be shown in this paper. Other functions included are: sin, cos, exp, square root, division, mix. The function mix(a, b, c, d) is a function which blends expressions c and d depending on the parameters a and b. We show an example of an expression tree of depth 5 in figure 4, along with the corresponding image. For the other images in this paper, we used a depth of 12.

Publius: A robust, tamper-evident, censorship-resistant web publishing system

Marc Waldman
Computer Science Dept.
New York University
waldman@cs.nyu.edu

Aviel D. Rubin
AT&T Labs–Research
rubin@research.att.com

Lorrie Faith Cranor
AT&T Labs–Research
lorrie@research.att.com

Abstract

We describe a system that we have designed and implemented for publishing content on the web. Our publishing scheme has the property that it is very difficult for any adversary to censor or modify the content. In addition, the identity of the publisher is protected once the content is posted. Our system differs from others in that we provide tools for updating or deleting the published content, and users can browse the content in the normal point and click manner using a standard web browser and a client-side proxy that we provide. All of our code is freely available.

1 Introduction

The publication of written words has long been a tool for spreading new (and sometimes controversial) ideas, often with the goal of bringing about social change. Thus the printing press, and more recently, the World Wide Web, are powerful revolutionary tools. But those who seek to suppress revolutions possess powerful tools of their own. These tools give them the ability to stop publication, destroy published materials, or prevent the distribution of publications. And even if they cannot successfully censor the publication, they may intimidate and physically or financially harm the author or publisher in order to send a message to other would-be-revolutionaries that they would be well advised to consider an alternative occupation. Even without a threat of personal harm, authors may wish to publish their works anonymously or pseudonymously because they believe they will be more readily accepted if not associated with a person of their gender, race, ethnic background, or other characteristics.

Quotations about the Internet's ability to resist censorship and promote anonymity have become nearly cliche. John Gillmore's quote "The Net treats censorship as damage and routes around it" has been interpreted as a statement that the Internet cannot be censored. And Peter Steiner's famous New Yorker cartoon captioned "On the Internet, nobody knows you're a dog" has been used to hype the Internet as a haven of anonymity. But increasingly people have come to learn that unless they take extraordinary precautions, their online writings can be censored and the true identity behind their online psuedonyms can be revealed.

Examples of the Internet's limited ability to resist censorship can be found in the Church of Scientology's attempts to stop the online publication of documents critical of the Church. Since 1994 the Church has combed the Internet for documents that contain what they describe as Church secrets. Individual authors, Internet service providers, and major newspapers such as *The Washington Post*, have had to defend their publication of excerpts from Church documents (some of them fewer than 50 words) in court. The Church has used copyright and trademark law, intimidation, and illegal searches and seizures in an attempt to suppress the publication of Church documents [13]. In 1995 the Church convinced the Finnish police to force Julf Helsingius, the operator of anonymous remailer anon.penet.fi, to reveal the true name of a user who had made anonymous postings about the Church. When the Church tried to obtain the names of two more users the following year, Helsingius decided to shut the remailer down [16].

The U.S. Digital Millennium Copyright Act, established to help copyright owners better protect their intellectual property in an online environment, is also proving to be yet another useful tool for censors. The Act requires online service providers to take down content upon notification from a copyright owner that the content infringes their copyright. While there is a process in place for the content owner to refute the infringement claim, the DMCA requires the online service provider to take down the content immediately and only restore it later if the infringement claim is

not proven to be valid.

We developed Publius in an attempt to provide a Web publishing system that would be highly resistant to censorship and provide publishers with a high degree of anonymity. Publius was the pen name used by the authors of the Federalist Papers, Alexander Hamilton, John Jay, and James Madison. This collection of 85 articles, published pseudonymously in New York State newspapers from October 1787 through May 1788, was influential in convincing New York voters to ratify the proposed United States constitution [17].

1.1 Design Goals

Nine design goals were important in shaping the design of Publius.

Censorship resistant Our system should make it extremely difficult for a third party to make changes to or force the deletion of published materials.

Tamper evident Our system should be able to detect unauthorized changes made to published materials.

Source anonymous There should be no way to tell who published the material once it is published on the web. (This requires an anonymous transport mechanism between publishers and web servers.)

Updateble Our system should allow publishers to make changes to their own materials or delete their own materials should they so choose.

Deniable Since our system relies on parties in addition to the publisher (as do most publishing systems, online and offline), those third parties should be able to deny knowledge of the content of what is published.

Fault tolerant Our system should still work even if some of the third parties involved are malicious or faulty.

Persistent Publishers should be able to publish materials indefinitely without setting an upfront expiration date.

Extensible Our system should be able to support the addition of new features as well as new participants.

Freely available All software required for our system should be freely available.

2 Related work

For the purposes of this paper, current Web anonymizing tools are placed into one of two categories. The first category consists of tools that attempt to provide connection based anonymity – that is the tool attempts to hide the identity of the individual requesting a particular Web page. The second category consists of tools that attempt to hide the location or author of a particular Web document. Although Publius falls into the latter category we briefly survey connection based anonymity tools as they can be used in conjunction with Publius to further protect an author's anonymity.

2.1 Connection Based Anonymity Tools

The Anonymizer (`http://www.anonymizer.com`) provides connection based anonymity by acting as a proxy for HTTP requests. An individual wishing to retrieve a Web page anonymously simply sends a request for that page to the Anonymizer. The Anonymizer then retrieves the page and sends it back to the individual that requested it.

LPWA [9], now known as Proxymate, is an anonymizing proxy that also offers a feature that can automatically generate unique pseudonymous user names (with corresponding passwords) and email addresses that users can send to Web sites. Every time a user returns to a particular Web site, the same pseudonyms are generated. The functionality of the anonymizing proxy is very similar to that of the Anonymizer.

Several anonymity tools have been developed around the concept of mix networks [5]. A mix network is a collection of routers, called mixes, that use a layered encryption technique to encode the path communications should take through the network. In addition, mix networks use other techniques such as buffering and message reordering to further obscure the correlation between messages entering and exiting the network.

Onion Routing [18] is a system for anonymous and private Internet connections based on mix networks. An Onion Routing user creates a layered data structure called an onion that specifies the encryption algorithms and keys to be used as data is transported to the intended recipient. As the data passes through each onion router along the way, one layer of encryption is removed according to the recipe contained in the onion. The request arrives at the recipient in plain text, with only the IP address of the last onion-router on the path. An HTTP proxy has been developed

that allows an individual to use the Onion Router to make anonymous HTTP requests.

Crowds [19] is an anonymity system based on the idea that people can be anonymous when they blend into a crowd. As with mix networks, Crowds users need not trust a single third party in order to maintain their anonymity. Crowds users submit their requests through a crowd, a group of Web surfers running the Crowds software. Crowds users forward HTTP requests to a randomly-selected member of their Crowd. Neither the end server nor any of the crowd members can determine where the request originated. The main difference between a mix network and Crowds is in the way paths are determined and packets are encrypted. In mix networks, packets are encrypted according to a pre-determined path before they are submitted to the network; in Crowds, a path is configured as a request traverses the network and each crowd member encrypts the request for the next member on the path. Crowds also utilizes efficient symmetric ciphers and was designed to perform much better than mix-based solutions.

The Freedom anonymity system (`http://www.freedom.net`) provides an anonymous Internet connection that is similar to Onion Routing; however, it is implemented at the IP layer rather than the application level. Freedom supports several protocols including HTTP, SMTP, POP3, USENET and IRC. In addition Freedom allows the creation of pseudonyms that can be used when interacting with Web sites or other network users.

2.2 Author Based Anonymity Tools

Janus, currently known as Rewebber (`http://www.rewebber.de`), is a combination author and connection based anonymizing tool. With respect to connection based anonymity, Janus functions almost exactly like the Anonymizer; it retrieves Web pages on an individual's behalf. Publisher anonymity is provided by a URL rewriting service. An individual submits a URL U to Janus and receives a Janus URL in return. A Janus URL has the following form

http://www.rewebber.com/surf-encrypted/$E_k(U)$

Where E_k(U) represents URL U encrypted with Janus's public key. This new URL hides U's true value and therefore may be used as an anonymous address for URL U. Upon receiving a request for a Janus URL, Janus simply decrypts the encrypted part of the URL with its private key. This reveals the Web page's true location to Janus. Janus now retrieves the page and sends it back to the requesting client. Just before Janus sends the page back to the client each URL, contained in the page, is converted into a Janus URL.

Goldberg and Wagner [12] describe their implementation of an anonymous Web publishing system based on a network of Rewebbers. The Rewebber network consists of a collection of networked computers, each of which runs an HTTP proxy server and possesses a public/private key pair. Each HTTP proxy server is addressable via a unique URL. An individual wishing to hide the true location of WWW accessible file f, first decides on a set of Rewebber servers through which a request for file f is to be routed. Using an encryption technique similar to the one used in onion routing, the URLs of these Rewebber servers are encrypted to form a URL U. Upon receiving an HTTP GET request for URL U, the Rewebber proxy uses its private key to *peel* away the outermost encryption layer of U. This decryption reveals only the identity of the next Rewebber server that the request should be passed to. Therefore only the last Rewebber server in the chain knows the true location of f. The problem with this scheme is that if any of the Rewebber servers along the route crashes, then file f cannot be found. Only the crashed file server possesses the private key that exposes the next server in the chain of Rewebber servers that eventually leads to file f. The use of multiple Rewebber servers and encryption leads to long URLs that cannot be easily memorized. In order to associate a meaningful name with these long URLs the TAZ server was invented. TAZ servers provide a mapping of names (ending in .taz) to URLs in the same way that a DNS server maps domain names to IP addresses. This anonymous publishing system is not currently operating as it was built as a "proof of concept" for a class project.

Most of the previous work in anonymous Web publishing has been done in the context of building a system to realize Anderson's Eternity Service [2]. The Eternity Service is a server based storage medium that is resistant to denial of service attacks and destruction of most participating file servers. An individual wishing to anonymously publish a document simply submits it to the Eternity Service along with an appropriate fee. The Eternity Service then copies the document onto a random subset of servers participating in the Eternity Service. Once submitted, a document cannot be removed from the Eternity Service. Therefore an author cannot be forced, even under threat, to delete a document published on the Eternity Service. Below we review several projects whose goals closely mirror or were inspired by the Eternity Service.

Usenet Eternity [3] is a Usenet news based implementation of a scaled down version of Anderson's Eternity Service. The system uses Usenet to store anonymously published documents. Documents to be published anonymously must be formatted according to a specific set of rules that call for the addition of headers and processing by PGP and SHA1. The correctly formatted message is then sent to alt.anonymous.messages. A piece of software called the eternity server is used to read the anonymously posted articles from the alt.anonymous.messages newsgroup. The eternity server is capable of caching some newsgroup articles. This helps prevent the loss of a document when it is deleted from Usenet. The problem with using Usenet news to store the anonymously published file is that an article usually exists on a news server for only a short period of time before it is deleted. In addition a posting can be censored by a particular news administrator or by someone posting *cancel* or *supercede* requests (`http://www.faqs.org/faqs/usenet/cancel-faq/`) to Usenet.
A much more ambitious implementation is currently being designed (`http://www.cypherspace.org/eternity-design.html`).

FreeNet [7] is an *adaptive network* approach to the censorship problem. FreeNet is composed of a network of computers (nodes) each of which is capable of storing files locally. In addition, each node in the network maintains a database that characterizes the files stored on some of the other nodes in the network. When a node receives a request for a non-local file it uses the information found in its database to decide which node to forward the request to. This forwarding is continued until either the document is found or the message is considered timed-out. If the document is found it is passed back through the chain of forwarding nodes. Each node in this chain can cache the file locally. It is this caching that plays the main role in dealing with the censorship issue. The multiple copies make it difficult for someone to censor the material. A file can be published anonymously by simply uploading it to one of the nodes in the adaptive network. The FreeNet implementation is still in its infancy and many features still need to be implemented.

Intermemory [11] is a system for achieving an immense self-replicating distributed persistent RAM using a set of networked computers. An individual wishing to join the Intermemory donates some disk space, for an extended period of time, in exchange for the right to store a much smaller amount of data in the Intermemory. Each donation of disk space is incorporated into the Intermemory. Data stored on the Intermemory is automatically replicated and dispersed. It is this replication and dispersion that gives the Intermemory properties similar to Anderson's Eternity Service. The main focus of the Intermemory project is not anonymous publishing but rather the preservation of electronic media. A small Intermemory prototype is described in [6]. The security and cryptographic components were not fully specified in either paper so we cannot comment on its anonymity properties.

Benes [4] describes in detail how one might implement a full-fledged Eternity service. Benes and several students at Charles University are attempting to create a software implementation of the Eternity Service based on this thesis.

3 Publius

In this section we describe how our system achieves the stated goals. We call the content that is published with the desired robustness properties *Publius content.*

3.1 Overview

Our system consists of *publishers* who post Publius content to the web, *servers* who host random-looking content, and *retrievers* who browse Publius content on the web. At present the system supports any static content such as HTML pages, images, and other files such as postscript, pdf, etc. Javascript also works. However, there is no support for interactive scripting such as CGI. Also, Java applets on Publius pages are limited in what they can do.

We assume that there is a static, system-wide list of available servers. Publius content is encrypted by the publisher and spread over some of the web servers. In our current system, the set of servers is static. The publisher takes the key, K that is used to encrypt the file to be published and splits it into n shares, such that any k of them can reproduce the original K, but $k-1$ give no hints as to the key [22].

Each server receives the encrypted Publius content and one of the shares. At this point, the server has no idea what it is hosting – it simply stores some random looking data.

To browse content, a retriever must get the encrypted Publius content from some server and k of the shares. As described below, a mechanism is in place to detect if the content has been tampered with. The publishing process produces a special URL that is used to recover the data and the shares. The published content is cryptographically tied to the URL.

Any modification to the stored Publius content or the URL results in a failed tamper check. If all tamper checks fail the Publius content cannot be read.

In addition to the publishing mechanism, we provide a way for publishers (and nobody else) to update or delete their Publius content. In the next several sections, we describe the Publius functions in some detail. We use a simple example of a publisher with one HTML file. Publishing more complicated content, such as web pages that have links to each other, is covered in Section 4.

3.2 Publish

The following text describes the publish pseudocode of Figure 1. This pseudocode is executed by the Publius client proxy in response to a publish request. To publish Publius content, M, the publisher, Alice, first generates a random symmetric key, K. She then encrypts M to produce $\{M\}_K$, M encrypted under K, using a strong symmetric cipher. Next, Alice splits K into n shares using Shamir secret sharing, such that any k of them can reproduce the secret.

For each of the n shares, Alice computes

$$name_i = wrap(H(M \cdot share_i))$$

That is, each share has a corresponding name. The name is calculated by concatenating the share with the message, taking a cryptographic hash, H, of the two, and xoring the first half of the hash output with the second half. We call the xor of the two halves *wrap*. In our system, we use MD5 [20] as the hash function, so each $name_i$ is 8 bytes long. Note that the $name_i$'s are dependent on every bit of the web page contents and the share contents. The $name_i$ values are used in the Publius server addressing scheme described below.

Recall that each publisher possesses a static list of size m of the available servers in the system. For each of the n shares, we compute

$$location_i = (name_i \text{ MOD } m) + 1$$

to obtain n values each between 1 and m. If at least d unique values are not obtained, we start over and pick another K. The value d represents the minimum number of unique servers that will hold the Publius content. Clearly this value needs to be greater than or equal to k since at least k shares are needed to reconstruct the key K. d should be somewhat smaller than m. It is clearly desirable to reduce the number of times we need to generate a new key K. Therefore we need to create a sufficient number of shares so that, with high probability, d unique servers are found. This problem is equivalent to the well known Coupon Collectors Problem [15]. In the Coupon Collectors Problem there are y different coupons that a collector wishes to collect. The collector obtains coupons one at a time, randomly with repetitions. The expected number of coupons the collector needs to collect before obtaining all y different coupons is $y * \ln(y)$. By analogy, a unique slot in the available server list is equivalent to a coupon. Therefore for each key K we create $n = \lceil d * \ln(d) \rceil$ shares. Any unused shares are thrown away.

Now, Alice uses each $location_i$ as an index into the list of servers. Alice publishes $\{M\}_k$, $share_i$, and some other information in a directory called $name_i$ on the server at location $location_i$ in the static list. Thus, given M, K, and m, the locations of all of the shares are uniquely determined. The URL that is produced contains at least d $name_i$ values concatenated together. A detailed description of the URL structure is given in Section 4.

Figure 2 illustrates the publication process.

3.3 Retrieve

The following text describes the retrieve pseudocode of Figure 3. This pseudocode is executed by the Publius client proxy in response to a retrieve request. The retriever, Bob, wishes to view the Publius content addressed by Publius URL U. Bob parses out the $name_i$ values from U and for each one computes

$$location_i = (name_i \text{ MOD } m) + 1$$

Thus, he discovers the index into the table of servers for each of the shares. Next, Bob chooses k of these arbitrarily. From this list of k servers, he chooses one and issues an HTTP GET command to retrieve the encrypted file and the share. Bob knows that the encrypted file, $\{M\}_K$ is stored in a file called *file* on each server, in the $name_i$ directory. The key share is stored in a file called *share* in that same directory.

Next, Bob retrieves the other $k - 1$ shares in a similar fashion (If all goes well, he does not need to retrieve any other files or shares). Once Bob has all of the shares, he combines them to form the key, K. Then, he decrypts the file. Next, Bob verifies that all of the $name_i$ values corresponding to the selected shares are correct by recomputing

$$name_i = wrap(H(M \cdot share_i))$$

using M that was just decrypted. If the k $name_i$'s are all correct (i.e. if they match the ones in the URL),

```
Procedure Publish (document M)
        Generate symmetric key K
        Encrypt M under key K producing {M}_K
        Split key K into n shares such that k shares are required to reconstruct K
        Store the n shares in array share[1..n]
        locations_used = {}
        for i = 1 to n:
                name=MD5(M · share[i])
                name=XOR(top_64_bits(name),bottom_64_bits(name))
                location=(name MOD serverListSize)+1
                if (location is not a member of locations_used):
                    locations_used = locations_used ∪ {location}
                    serverIP_Address = serverList[location]
                    Insert (serverIP_Address, share[i]) into Publish_Queue
                    publiusURL = publiusURL · name
                endif
        endfor
        if (sizeof(locations_used)< d) then
                Empty (Publish_Queue)
                return Publish(M)
        else
                for each (serverIP_Address, share) in Publish_Queue:
                    HTTP_PUT({M}_K and share on Publius Server with IP address serverIP_Address)
                    return publiusURL
        endif
End Publish
```

Figure 1: Publish Algorithm

Bob can be satisfied that either the document is intact, or that someone has found a collision in the hash function.

If something goes wrong, Bob can try a different set of k shares and an encrypted file stored on one of the other n servers. In the worst case, Bob may have to try all of the possible $\binom{n}{k}$ combinations to get the web page before giving up. An alternate retrieval strategy would be to try all $n* \binom{n}{k}$ combinations of shares and documents. Each encrypted document can be tested against each of the $\binom{n}{k}$ share combinations.

If we are willing to initially download all the shares from all the servers then yet another method for determining the key becomes available. In [10], Gemmell and Sudan present the Berlekamp and Welch method for finding the polynomial, and hence the key K, corresponding to n shares of which at most j are corrupt. The value j must be less than $(n-d)/2$ where d is one less than the number of shares needed to form the key. However if the number of corrupt shares is greater than $(n-d)/2$ we are not quite out of luck. We can easily discover whether K is incorrect by performing the verification step described above. Once we suspect that key K is incorrect we can just perform a brute force search by trying all $n* \binom{n}{k}$ combinations of shares and documents. The following example illustrates this point. If we have $n = 10$ shares and require 3 shares to form K then the Berlekamp and Welch method will generate the correct polynomial only if less than $((10-2)/2) = 4$ shares are corrupted. Suppose 6 shares are corrupt. Of course we don't know this ahead of time so we perform the Berlekamp and Welch method which leads us to key K. Key K is tested against a subset of, or perhaps all, the encrypted documents. All of the tamper check failures lead us to suspect that K is incorrect. Therefore we perform a brute force search for the correct key by trying all $n* \binom{n}{k}$ combinations of shares and documents. Assuming we have a least one untampered encrypted document this method will clearly succeed as we have 4 uncorrupted shares, only three of which are needed to form the correct key.

Once the web page is retrieved Bob can view it in his browser. In our implementation, all of the work is handled by the proxy. Publius URLs are tagged as special, and they are parsed and handled in the proxy. The proxy retrieves the page, does all of the verification, and returns the web content to the browser. So, all of this is transparent to the user. The user just points and clicks as usual. Section 4 describes Publius URL's and the proxy software in detail.

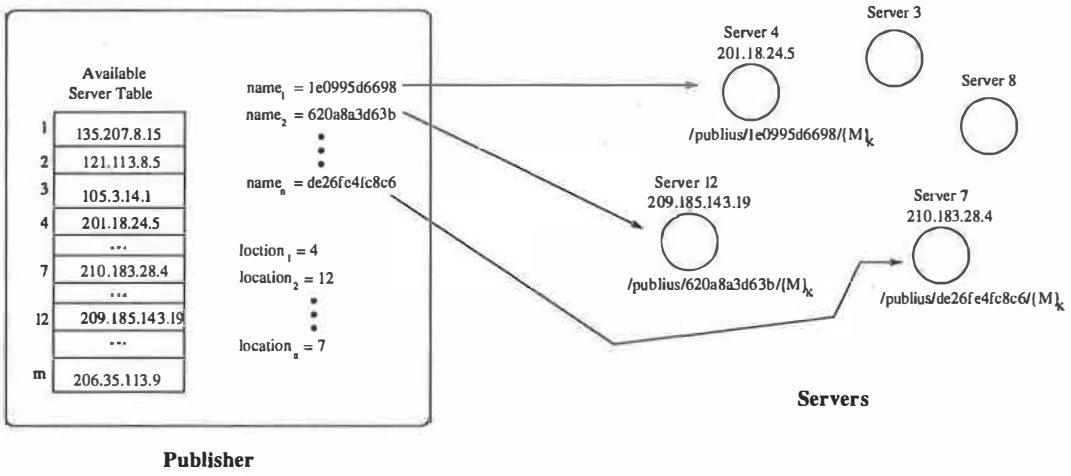


Figure 2: **The Publius publication process** The publisher computes the $name_i$ values by hashing the web page and the symmetric key shares together. Then, those values are used to compute the locations. The publisher then uses the location value as an index into the static location table and publishes the encrypted file, along with the share in a directory named $name_i$ on the appropriate server.

3.4 Delete

It is desirable for Alice to be able to delete her Publius content from all servers, while nobody else should be able to delete this content. To achieve this, just before Alice publishes a file she generates a password PW. Alice then sends the encrypted document, share and $H(server\_domain\_name \cdot PW)$ to the servers that will be hosting Alice's published document. $H(server\_domain\_name \cdot PW)$ is the hash of the domain name of the server concatenated with a password PW. The server stores this hash value in the same directory as the encrypted file and the share, in a file called *password*. The reason this value is stored as opposed to just the PW or H(PW), is that it prevents a malicious server from learning the password and deleting the associated Publius content from all other servers that are hosting it.

We implemented *delete* as a CGI script running on each server. To delete Publius content, Alice sends $H(server\_domain\_name \cdot PW)$ to each hosting server, along with the $name_i$ that corresponds to the that server. The server compares the password received to the one stored, and if they match, removes the directory matching the $name_i$, and all of the files in it.

3.5 Update

Our system provides a mechanism for Alice to update something that she previously published. We use the same password mechanism that is used to delete content. Thus, Alice can change any web page that she published, but nobody else can. The idea is to enable Alice to change content without changing the URL, because others may have linked to the original site. After the update, anyone retrieving the original URL receives the new content.

In addition to *file*, *share*, and *password*, there is a file called *update* on the servers in the $name_i$ directory. Initially, if Alice has not updated the content, the file does not exist. When Bob retrieves the URL, if the update file is missing, everything proceeds as described in Section 3.3.

To update the content, Alice specifies a file name containing the new content, the original URL, the original password PW and a new password. The update program first publishes the new content by simply calling publish with the file name and the new password. Once the new content is published, the original URL is used to find the n servers that host the Publius content. Each of these servers receives a message from Alice (a call to a CGI script) containing the original password stored on that server (recall that this is $H(server \cdot PW)$), the old $name_i$, and the new URL. Each server then places the new URL in the *update* file and deletes the contents in the old *file*.

When Bob retrieves Publius content, if the update file exists, the servers return the update URL instead of the contents. Bob receives the update URL from k servers and compares them, if they are all equal, he then retrieves the new URL instead. Of course, Bob is not aware of what the retrieve program is doing behind the scenes. From his point of view, he makes

```
Procedure Retrieve (PubliusURL U)
        // k is the number of shares needed to reconstruct Key K
        // n is the number of name[i] values stored in the PubliusURL U
        // URL U = name[1]...name[n]
        S={set of all unique k-subsets of the elements (1..n)}
        for each element s in S:
                R=randomValue(k) // choose a random integer in range 1..k
                for i = 1 to k:
                    v = i^th element of set s
                    location=(name[v] MOD serverListSize)+1
                    serverIP_Address=serverList[location]
                    share[i]=retrieve file "share" from server at serverIP_Address
                    tamperCheckValue[i]=name[v]
                    if (i==R) then
                        encryptedDocument=retrieve {M}_k from server at serverIP_Address
                    endif
                endfor
                K=reconstructKeyFromShares(share[1]...share[k])
                M=Decrypt encryptedDocument using key K
                tamperCheckPassed=TRUE
                for i = 1 to k:
                    V=MD5(M . share[i])
                    V=XOR(top_64_bits(V),bottom_64_bits(V))
                    if (V ≠ tamperCheckValue[i]) then
                        tamperCheckPassed=FALSE
                        break
                    endif
                endfor
                if (tamperCheckPassed) then
                    return M
                endif
        endfor
        return "Document cannot be retrieved"
End Retrieve
```

Figure 3: Retrieve Algorithm

a request and receives the web page. If the k URLs do not match, Bob (his proxy) then tries the other $n-k$ servers until he either gets k that are the same, or gives up. In Section 5 we discuss other ways this could be implemented and several tradeoffs that arise.

Although the update mechanism is very convenient it leaves Publius content vulnerable to a redirection attack. In this attack several malicious server administrators collaborate to insert an update file in order to redirect requests for the Publius content. A mechanism exists within Publius to prevent such an attack. During the publicaton process the publisher has the option of declaring a Publius URL as nonupdateable. When a Publius client attempts to retrieve Publius content from a nonupdateable URL all update URLs are ignored. See Section 4.1 for more information about nonupdateable URLs.

4 Implementation issues

In this section we describe the software components of Publius and how these components implement Publius functions.

4.1 Publius URLs

Each successfully published document is assigned a Publius URL. A Publius URL has the following form

$$http://!anon!/options\ encode(name_1)\ldots encode(name_n)$$

where $name_i$ is defined as in Section 3.2 and the encode function is the Base64 encoding function (verb+http://www.ietf.org/rfc/rfc1521.txt+). The Base64 encoding function generates an ASCII representation of the $name_i$ value.

The *options* section of the Publius URL is made up of 2 characters that define how the Publius client

software interprets the URL. This 16 bit *options* section encodes three fields – the version number, the number of shares needed to form a key, and finally the update flag. The version number allows us to add new features to future versions of Publius while at the same time retaining backward compatibility. It also allows Publius clients to warn a user if a particular URL was meant to be interpreted by a different version of the client software. The next field identifies the number of shares needed to form the key K. The last field is the *update* flag that determines whether or not the update operation can be performed on the Publius content represented by the URL. If the *update* flag is a 1 then the retrieval of updated content will be performed in the manner described in Section 3.5. However if the *update* flag is 0 then the client will ignore update URLs sent by Publius servers in response to share and encrypted file requests. The update flag's role in preventing certain types of attacks is described in Section 5.

Many older browsers enforce the rule that a URL can contain a maximum of 256 characters. The initial "http://!anon!/" string is 14 characters long, leaving 242 characters for the 20 $name_i$ values. Base64 processes data in 24 bit blocks, producing 4 ASCII characters per 24 bit block. This results in 12 ASCII characters per $name_i$ value. Twenty hashes produce 240 ASCII characters. Thus, older browsers restrict us to 20 different publishing servers in our scheme. We use the two remaining characters for the *options* section described above.

Here is an example of a Publius URL:

http://!anon!/AH2LyMOBWJrDw=
GTEaS2GlNNE=NIBsZlvUQP4=sVfdKF7o/kl=
EfUTWGQU7LX=OCk7tkhWTUe=GzWiJyio75b=
QUiNhQWyUW2=fZAX/MJnq67=y4enf3cLK/0=

4.2 Server software

To participate as a Publius server, one only needs to install a CGI script that we provide. All client software communicates with the server by executing an HTTP POST operation on the server's CGI URL. The requested operation (*retrieve*, *update*, *publish* or *delete*), the file name, the password and any other required information is passed to the server in the body of the POST request. We recommend limiting the amount of disk space that can be used each time the CGI script executes. Our CGI script is freely available (see Section 7).

4.3 Client software

The client software consists of an HTTP proxy and a set of publishing tools. An individual wishing only to retrieve Publius content just requires the proxy. The proxy transparently sends non Publius URLs to the appropriate servers and passes the returned content back to the browser. Upon receiving a request for a Publius URL the proxy first retrieves the encrypted document and shares as described in Section 3.3 and then takes one of three actions. If the decrypted document successfully verifies, it is sent back to the browser. If the proxy is unable to find a document that successfully verifies an HTML based error message is returned to the browser. If the requested document is found to have been updated then an HTTP redirect request is sent to the browser along with the update URL.

4.4 Publishing mutually hyperlinked documents

Suppose Alice wants to anonymously publish HTML files A and B. Assume that file A contains a hyperlink to file B. Alice would like the anonymously published file A to retain its hyperlink to the anonymously published file B. To accomplish this, Alice first publishes file B. This action generates a Publius URL for file B, B_{url}. Alice records B_{url} in the appropriate location in file A. Now Alice publishes file A. Her task is complete.

Alice now wishes to anonymously publish HTML files C and D. File C has a hyperlink to file D and file D has a hyperlink to file C. Alice now faces the dilemma of having to decide which file to publish first. If Alice publishes file C first then she can change D's hyperlink to C but she cannot change C's hyperlink to D because C has already been published. A similar problem occurs if Alice first publishes file D.

The problem for Alice is that the content of a file is cryptographically tied to its Publius URL – changing the file in any way changes its Publius URL. This coupled with the fact that file C and file D contain hyperlinks to each other generates a circular dependency – each file's Publius URL depends on the other's Publius URL. What is needed to overcome this problem is a way to break the dependency of the Publius URL on the file's content. This can be accomplished using the Publius Update mechanism described in Section 3.5.

Using the update mechanism Alice can easily solve the problem of mutually hyperlinked files. First Alice publishes files C and D in any order. This generates Publius URL C_{url} for file C and Publius URL D_{url}

for file D. Alice now edits file C and changes the address of the D hyperlink to D_{url}. She does the same for file D – she changes the address of the C hyperlink to C_{url}. Now she performs the Publius Update operation on C_{url} and the newly modified file C. The same is done for D_{url} and the newly updated file D. This generates Publius URL C_{url_2} for for file C and Publius URL D_{url_2} for file D. The problem is solved. Suppose Bob attempts to retrieve file C with C_{url}. Bob's proxy notices the file has been updated and retrieves the file from C_{url_2}. Some time later, Bob clicks on the D hyperlink. Bob's proxy requests the document at D_{url} and is redirected to D_{url_2}. The update mechanism ensures that Bob reads the latest version of each document.

4.5 Publishing a directory

Publius contains a directory publishing tool that automatically publishes all files in a directory. In addition, if some file, *f*, contains a hyperlink to another file, *g*, in that same directory, then *f's* hyperlink to *g* is rewritten to reflect *g's* Publius URL. Mutually hyperlinked HTML documents are also dealt with, as described in the previous section.

The first step in publishing a directory, *D*, is to publish all of *D*'s non-HTML files and record, for later use, each file's corresponding Publius URL. All HTML files in *D* are then scanned for hyperlinks to other files within *D*. If a hyperlink, *h*, to a previously published non-HTML file, *f*, is found then hyperlink *h* is changed to the Publius URL of *f*. Information concerning hyperlinks between HTML files in directory *D* is recorded in a data structure called a dependency graph. Dependency graph, *G*, is a directed graph containing one node for each HTML file in *D*. A directed edge *(x,y)* is added to *G* if the HTML file *x* must be published before file *y*. In other words, the edge *(x,y)* is added if file *y* contains a hyperlink to file *x*. If, in addition, file *x* contains a hyperlink to file *y* the edge *(y,x)* would be added to the graph causing the creation of a cycle. Cycles in the graph indicate that we need to utilize the Publius Update trick that Alice uses when publishing her mutually hyperlinked files C and D (Section 4.4).

Once all the HTML files have been scanned the dependency graph *G* is checked for cycles. All HTML files involved in a cycle are published and their Publius URLs recorded for later use. Any hyperlink, *h*, referring to a file, *f*, involved in a cycle, is replaced with *f's* Publius URL. All nodes in the cycle are removed from *G* leaving *G* cycle-free. A topological sort is then performed on *G* yielding *R*, the publishing order of the remaining HTML files. The result of a topological sort of a directed acyclic graph (DAG) is a linear ordering of the nodes of the DAG such that if there is a directed edge from vertex *i* to vertex *j* then *i* appears before *j* in the linear ordering [1]. The HTML files are published according to order *R*. After each file, *f*, is published, all hyperlinks pointing to *f* are modified to reflect *f's* Publius URL. Finally a Publius Update operation is performed on all files that were part of a cycle in *G*.

4.6 Publius content type

The file name extension of a particular file usually determines the way in which a Web browser interprets the file's content. For example, a file that has a name ending with the extension ".htm" usually contains HTML. Similarly a file that has a name ending with the extension ".jpg" usually contains a JPEG image. The Publius URL does not retain the file extension of the file it represents. Therefore the Publius URL gives no hint to the browser, or anyone else for that matter, as to the type of file it points to. Indeed, this is the desired behavior as we do not wish to give the hosting server the slightest hint as to the type of content being hosted. However, in order for the browser to correctly interpret the byte stream sent to it by the proxy, the proxy must properly identify the type of data it is sending. Therefore before publishing a file we prepend the first three letters of the file's name extension to the file. We prepend the three letter file extension rather than the actual MIME type because MIME types are of variable length (An alternative implementation could store the actual MIME type prepended with two characters that represented the length of the MIME type string). The file is then published as described in Section 3.2. When the proxy is ready to send the requested file back to the browser the three letter extension is removed from the file. This three letter extension is used by the proxy to determine an appropriate MIME type for the document. The MIME type is sent in an HTTP "Content-type" header. If the three letter extension is not helpful in determining the MIME type a default type of "text/plain" is sent for text files. The default MIME type for binary files is "octet/stream".

4.7 User interface

The client side software includes command line tools to perform the publish, delete and update operations described in section 3. The retrieve operation is performed via the Web browser in conjunction with the proxy. In addition, a Web browser based interface to the tools has been developed. This browser based

interface allows someone to select the Publius operation (*retrieve*, *update*, *publish* or *delete*) and enter the operation's required parameters such as the URL and password. Each Publius operation is bound to a special !anon! URL that is recognized by the proxy. For example the publish URL is !anon!PUBLISH. The operation's parameters are sent in the body of the HTTP POST request to the corresponding !anon! URL. The proxy parses the parameters and executes the corresponding Publius operation. An HTML message indicating the success or failure of the operation is returned. If the retrieve operation is requested, and is successful, the requested document is displayed in a new Web browser window.

5 Limitations and threats

In this section we discuss the limitations of Publius and how these limitations could be used by an adversary to censor a published document, disrupt normal Publius operation, or learn the identity of an author of a particular document. Possible countermeasures for some of these attacks are also discussed.

5.1 Share deletion or corruption

As described in section 3.2, when a document is successfully published a copy of the encrypted document and a share are stored on each of the n servers. Only one copy of the encrypted document and k shares are required to recover the original document.

Clearly, if all n copies of the encrypted file are deleted, corrupted or otherwise unretrievable then it is impossible to recover the original document. Similarly if n-k+1 shares are deleted, corrupted or cannot be retrieved it is impossible to recover the key. In either case the published document is effectively censored. This naturally leads to the conclusion that the more we increase n, or decrease k, the harder we make it for an individual, or group of individuals, to censor a published document.

5.2 Update file deletion or corruption

As stated in section 3.5, if a server receives a request for Publius content that has an associated update file, the URL contained in that file is sent back to the requesting proxy.

We now describe three different attacks on the update file that could be used by an adversary to censor a published document. In each of these attacks the adversary, Mallory, has read/write access to all files on a server hosting the Publius content P, he wishes to censor.

In the first attack we describe, P does not have an associated update file. That is, the author of P has not executed the Publius Update operation on P's URL. Mallory could delete P from one server, but this does not censor the content because there are other servers available. Rather than censor the Publius content, Mallory would like to cause any request for P to result in retrieval of a different document, Q, of his choosing. The Publius URL of Q is Q_{url}. Mallory now enters Q_{url} into a file called "update" and places that file in the directory associated with P. Now whenever a request for P is received by Mallory's server, Q_{url} is sent back. Of course Mallory realizes that a single Q_{url} received by the client does not fool it into retrieving Q_{url}. Therefore Mallory enlists the help of several other Publius servers that store P. Mallory's friends also place Q_{url} into an "update" file in P's directory. Mallory's censorship clearly succeeds if he can get an update file placed on every server holding P. If the implementation of Publius only requires that k shares be downloaded, then Mallory does not necessarily need to be that thorough. When the proxy makes a request for P, if Mallory is lucky, then k matching URLs are returned and the proxy issues a browser redirect to that URL. If this happens Mallory has censored P and has replaced it with Publius Content of his own creation. This motivates higher values for k. The *update* flag described in section 4.1 is an attempt to combat this attack. If the publisher turned the update flag off when the content was published then the Publius client interpreting the URL will refuse to accept the update URLs for the document. Although the content might now be considered to be censored, someone is not duped into believing that an updated file is the Publius content originally published.

In the second attack, P has been updated and there exists an associated update file containing a valid Publius URL that points to Publius Content U. To censor the content, Mallory must corrupt the update file on $n - k + 1$ servers. Now there is no way for anyone to retrieve the file correctly. In fact, if Mallory can corrupt that many servers, he can censor any document. This motivates higher values for n and lower values for k.

One other attack is worth mentioning. If Mallory can cause the update files on all of the servers accessed by the client to be deleted, then he can, in effect, restore Publius content to its previous state before the update occurred. This motivates requiring clients to retrieve from all n servers before performing verification.

The attacks described above shed light on a couple of tradeoffs. Requiring retrievers to download all n shares and n copies of the document is one extreme that favors censorship resistance over performance. Settling for only the first k shares opens the user up to a set of corrupt, collaborating servers. Picking higher values for k minimizes this problem. However, lower values of k require the adversary to corrupt more servers to censor documents. Thus, k, the number of shares, and the number of copies of the page actually retrieved, must be chosen with some consideration.

5.3 Denial of service attacks

Publius, like all Web services, is susceptible to denial of service attacks. An adversary could use Publius to publish content until the disk space on all servers is full. This could also affect other applications running on the same server. We take a simple measure of limiting each publishing command to 100K. A better approach would be to charge for space.

An interesting approach to this problem is a CPU cycle based payment scheme known as Hash Cash (`http://www.cypherspace.org/~adam/hashcash/`). The idea behind this system is to require the publisher to do some work before publishing. Thus, it becomes difficult to efficiently fill the server disk. Hopefully, the attack can be detected before the disk is full. In Hash Cash, a client wishing to store a file on a particular server first requests a challenge string c and a number, b, from that server. The client must find another string, s, such that at least b bits of $H(c \cdot s)$ match b bits of $H(s)$ where H is a secure hash function such as MD5 and "$\cdot$" is the concatenation operator. That is, the client must find partial collisions in the hash function.

The higher the value of b, the more time the client requires to find a matching string. The client then sends s to the server along with the file to be stored. The server only stores the file if $H(s)$ passes the b bit matching test on $H(c \cdot s)$. Another scheme we are considering is to limit, based on client IP address, the amount of data that a client can store on a particular Publius server within a certain period of time. While not perfect, this raises the bar a bit, and requires the attacker to exert more effort. We have not implemented either of these protection mechanisms yet.

Dwork and Naor in [8] describe several other CPU cycle based payment schemes.

5.4 Threats to publisher anonymity

Although Publius was designed as a tool for anonymous publishing there are several ways in which the identity of the publisher could be revealed.

Obviously if the publisher leaves any sort of identifying information in the published file he is no longer anonymous. Publius does not anonymize all hyperlinks in a published HTML file. Therefore if a published HTML page contains hyperlinks back to the publisher's Web server then the publisher's anonymity could be in jeopardy.

Publius by itself does not provide any sort of connection based anonymity. This means that an adversary eavesdropping on the network segment between the publisher and the Publius servers could determine the publisher's identity. If a server hosting Publius Content keeps a log of all incoming network connections then an adversary can simply examine the log to determine the publisher's IP address. To protect a publisher from these sort of attacks a connection based anonymity tool such as Crowds should be used in conjunction with Publius.

5.5 "Rubber-Hose cryptanalysis"

Unlike Anderson's Eternity Service [2] Publius allows a publisher to delete a previously published document. An individual wishing to delete a document published with Publius must possess the document's URL and password. An adversary who knows the publisher of a document can apply so called "Rubber-Hose" Cryptanalysis [21] (threats, torture, blackmail, etc) to either force the publisher to delete the document or reveal the document's password.

Of course the adversary could try to force the appropriate server administrators to delete the Publius Content he wants censored. However when Publius Content is distributed across servers located in different countries and/or jurisdictions such an attack can be very expensive or impractical.

6 Future Work

Most of the browsers and proxies in use today do not impose the 256 character limit on URL size. With this limit lifted a fixed table of servers is no longer needed as the Publius URL itself can contain the IP addresses of the servers on which the content is stored. With no predefined URL size limit there is essentially no limit to the number of hosting servers that can be stored in the Publius URL. The Publius URL structure remains essentially the same – just the IP addresses are added. The *option* and $name_i$ components of the URL remain as they are still needed for tamper checking and URL interpretation. We intend to use this URL format in future versions of Publius.

During the Publius publication process the encrypted file, along with other information, is stored on the host servers. Krawczyk in [14] describes how to use Rabin's information dispersal algorithm to reduce the size of the encrypted file stored on the host server. We are planning to use this technique to reduce amount of storage needed on host servers.

7 Conclusions and availability

In this paper we have described Publius, a Web based anonymous publishing system that is resistant to censorship. Publius's main contributions beyond previous anonymous publishing systems include an automatic tamper checking mechanism, a method for updating or deleting anonymously published material, and methods for anonymously publishing mutually hyperlinked content.

The current implementation of Publius consists of approximately fifteen hundred lines of Perl. The source code is freely available at `http://www.cs.nyu.edu/~waldman/publius.html`.

Acknowledgements

We would like to thank Usenix for supporting this work. We would also like to thank Adam Back, Ian Goldberg, Oscar Hernandez, Graydon Hoare, Benny Pinkus, Adam Shostack, Anton Stiglic, Alex Taler and the anonymous reviewers for their helpful comments and recommendations.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures And Algorithms.* Addison-Wesley Publishing Company, 1983.

[2] R. J. Anderson. The eternity service. In *Pragocrypt 1996*, 1996. http://www.cl.cam.ac.uk/users/rja14/eternity/eternity.html.

[3] A. Back. The eternity service. *Phrack Magazine*, 7(51), 1997. http://www.cypherspace.org/ adam/eternity/phrack.html.

[4] T. Benes. The eternity service. 1998. http://www.kolej.mff.cuni.cz/eternity/.

[5] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.

[6] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. N. Yianilos. A prototype implementation of archival intermemory. In *Proc. ACM Digital Libraries.* ACM, August 1999. http://www.intermemory.org/.

[7] I. Clark. A distributed decentralised information storage and retrieval system. 1999. http://freenet.sourceforge.net/Freenet.ps.

[8] C. Dwork and M.Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology—CRYPTO '92*, pages 139–147. Springer-Verlag, 1992.

[9] E.G. Gabber, P.B. Gibbons, D.M. Kristol, Y. Matias, and A. Mayer. Consistent, yet anonymous web access with LPWA. *Communications of the ACM*, 42(2):42–47, 1999.

[10] P. Gemmell and M. Sudan. Highly resilient correctors for polynomials. *Information Processing Letters*, 43:169–174, 1992.

[11] A. V. Goldberg and P. N. Yianilos. Towards and archival intermemory. In *Proc. IEEE International Forum on Research and Technology Advances in Digital Libraries (ADL'98)*, pages 147–156. IEEE Computer Society, April 1998. http://www.intermemory.org/.

[12] I. Goldberg and D. Wagner. TAZ servers and the rewebber network: Enabling anonymous publishing
on the world wide web. *First Monday*, 3, 1998. http://www.firstmonday.dk/issues/issue3_4/goldberg/index.html.

[13] Wendy M. Grossman. *Wired*, 3(12):172 –177 and 248–252, December 1995. http://www.wired.com/wired/archive/3.12/alt.scientology.war_pr.html.

[14] H. Krawczyk. Secret sharing made short. In *Advances in Cryptology—CRYPTO '93*, pages 136–143. Springer-Verlag, 1993.

[15] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[16] Ron Newman. The church of scientology vs. the net. February 1998. http://www2.thecia.net/users/rnewman/scientology/home.html.

[17] U.S. Library of Congress. About the federalist papers. http://lcweb2.loc.gov/const/fed/abt_fedpapers.html.

[18] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Proxies for anonymous routing. In *12th Annual Computer Security Applications Conference*, 1996. http://www.onion-router.net/Publications.html.

[19] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information System Security*, 1(1), April 1998.

[20] R. Rivest. The MD5 message digest algorithm. *RFC 1321*, April 1992.

[21] B. Schneier. *Applied Cryptography.* John Wiley and Sons, 1996.

[22] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, November 1979.

Probabilistic Counting of Large Digital Signature Collections

Markus G. Kuhn*

University of Cambridge
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG
United Kingdom
mgk25@cl.cam.ac.uk

Abstract

A large number of people digitally sign the same document. The signature collectors want to use only a small amount of memory to demonstrate to any third party approximately how many persons have signed it. The scheme described in this paper uses a non-uniform secure hash function to select a small subset of signatures that the collectors store. The size of this subset becomes a verifiable estimate for the logarithm of the number of signers.

Applications for this scheme range from Web page metering, through ranking mechanisms for electronic discussion systems, to the distributed verifiable and scalable delegation of power in jointly-administrated systems.

1 Introduction

Signature collection has a long tradition as a means of documenting public support and plays an important rôle in many political systems. Candidates for political positions often have to collect signatures from voters as an entry qualification for an election. Some constitutions allow voters to initiate a vote by collecting a sufficient number of signatures. Civil rights organizations frequently use signature collection to demonstrate broad public support for their agendas.

Signature collection can be a valuable mechanism for decision-making arrangements that involve a large number of participants. It differs in important aspects from voting, which can consume significant up-front resources, since every entitled voter has to be informed early enough about the opportunity to vote. The outcome of a vote only represents the distribution of opinions sufficiently well when enough voters participate.

In a signature collection on the other hand, the signers can only decide to either support a proposal by signing a prepared document or to abstain from the procedure entirely. The result of a signature collection thus explicitly indicates only a lower bound for the absolute number of supporters for the proposal and it provides no useful information on how many people rejected it. Signature collection therefore makes sense only in situations where it is in the interest of the originators of the proposal—and those who collect signatures for it—to get as many signatures as possible.

Signature collection can be performed in a meaningful way without activating potentially expensive mechanisms for inviting every entitled participant. While voting requires an agreed official channel for announcing the start of the procedure and the details of the choices, in a signature collection, the responsibility for any necessary announcements can be left to the initiators of a proposal.

A centralized voting announcement mechanism could become subject to frequent frivolous abuse. Its activation must therefore be protected from malicious participants, who could try to delay further votes by overusing their right to initiate one. Signature collection mechanisms on the other hand can be started locally without large-scale announcements. They therefore make a much less attractive target for denial-of-service attacks.

Signature collections can potentially become very large, involving many millions of signers. The goal of voting is to identify a majority choice and therefore every single vote could become critical in deciding the outcome. On the other hand, the goal of a signature collection is only to demonstrate that some significant number of participants supports a pro-

*Supported by a European Commission Marie Curie training grant

posal, and therefore a verifiable order-of-magnitude estimate for the number of signers is what matters rather than the precise count. The presentation of for example roughly a million supporting signatures can be an important demonstration of popularity of an idea, even if there is a 25% uncertainty with regard to the precise number of valid signatures.

The result of a conventional signature collection is presented by the collectors as boxes of handwritten paper lists. This form allows anyone to identify and verify a few selected sample signers, but it usually remains prohibitively expensive to create a database of all signers. Signed paper lists thus remain a widely accepted compromise between verifiability and privacy concerns.

In this paper, we propose a new technique for conducting the cryptographic equivalent of signature collection and outline some potential applications. Our technique features the following properties, the first three of which are shared with a handwritten signature collection:

- A small sample of the signers will become identifiable such that the correctness of the signature collection can be verified by contacting them. However, it is not necessary to provide a complete database of all signers in order to make the claimed number of supporting signatures verifiable.
- Only the order-of-magnitude number of valid signatures is practically verifiable via sampling and not the precise number.
- The result of a signature collection can be sent as a single protocol data unit to verifiers and no further interactions between the verifiers and the collectors will be necessary to complete the verification. This makes the scheme useful for store-and-forward delivery of the result.
- Many signature collectors can operate independently and can merge their results at a later time without risking multiple collection of the same signature by different collectors to influence the result.
- The verifiable result of a large-scale digital signature collection fits into a file not much larger than a few tens of kilobytes, which can be efficiently transmitted to and verified by many different parties.
- Signature collectors need only a small amount of storage capacity, just large enough to hold the compact final result that is provided to the verifiers.

2 Application Examples

In addition to the classical political applications, the digital equivalent of signature collection has also numerous interesting new uses in distributed systems. Some examples include:

2.1 Web Page Metering

Advertising companies sponsor the providers of free online content and services in exchange for presence on popular Web pages. The sponsors want evidence for the popularity of the points-of-visibility that they have purchased. A discussion of the requirements and some proposed protocols for Web page metering can be found in Naor and Pinkas [1].

If every user of the sponsored Web content automatically signed a *digital guest book* by submitting a message that confirms that this user was indeed interested in the presented content, then the content provider could easily prove the popularity of her pages to the sponsors. Compared to the secret-sharing based metering scheme suggested in [1], the signature collection scheme presented in the next section has the advantage that the approximate number of users does not have to be known in advance to select the metering parameters, and users do not have to fetch secret shares from a trusted third party before using a Web page. They only have to be part of a suitable public-key infrastructure, and the certificates do not even have to bind their web-metering keys to any cleartext name if their real identity must remain secret from both the content provider and the sponsor.

2.2 TV Rating and Opinion Polling

A closely related application field is counting the viewers of a TV channel or interactive opinion gathering via TV networks with return channels. Advertisement customers and other observers might not entirely trust the network operators to provide accurate viewer or voter counts.

Set-top boxes could implement a mechanism to allow TV viewers to participate in signature collections, the result of which could then be verified directly by anyone using the public keys of the set-top box (or conditional-access module) manufacturers, which would here also act as key certification authorities. Signature lists could be collected into a compact representation in local and regional network nodes, so as to avoid communication bottlenecks around a central collection point.

2.3 Newsgroup Contributor Ranking

In open electronic discussion forums such as Internet newsgroups or mailing lists, the quality of contributions varies greatly. Long-term participants in such forums could establish a cryptographic credibility record by collecting signatures from readers who considered their past contributions valuable. The size of these signature lists could be queried by new participants to get an initial estimate of the experience and signal-to-noise ratio of a contributor.

With the digital signature collection scheme presented here, such a ranking mechanism can be implemented without any additional trusted third party beyond the certification authorities that are required as part of any public-key infrastructure. If the signature collection result can be represented compactly, it can be distributed easily over the newsgroup server network, allowing every reader to verify the validity of the signature count efficiently and independently without relying on the integrity of distribution servers.

2.4 Delegation of Power and Joint Administration

It might be undesirable to have replicated repositories for public software and documents as well as discussion groups under the central control of a single organization. An interesting alternative would be a system design in which all participants can directly determine which individuals they would like to see authorized to perform administrative tasks, especially if there were an efficient way in which each storage server can independently verify such a direct authorization of an individual by a large user base.

Such systems could be set up so that owning a long list of recent supporting signatures from other participants would allow an individual directly to perform privileged operations like acting as a moderator with the right to clean-up inappropriate publications ("spam", insults, piracy, etc.). Here, the signature list becomes a way in which a large number of participants can delegate administrative power (usually with suitable time and scope restrictions) to individuals.

The same could be achieved with a voting mechanism. However, this would require common trust in some central voting server that determines and certifies the majority winners of this process. The compact representation of the verifiable result gives the signature collection approach robustness against the failure or compromise of other servers. Every server on which the administrative actions will be executed can independently and directly verify the public support for the moderators.

In all the previous applications, a trivial approach would be that the collector of signatures just stores all v collected signatures and hands them over to anyone who wants to verify that at least v signatures have indeed been collected. This would require $O(v)$ storage space, transmission time, and verification time, which can be prohibitive if the number v of collected signatures is large ($\gg 10^4$).

We therefore present a new technique that allows us to collect signatures and prove a probabilistic lower bound for their number to others using only $O(\log v)$ of storage space, transmission time, and verification effort.

3 A Probabilistic Representation of Signature Collections

The basic idea of this scheme is the representation of a complete collection of signatures by storing only a small set of sample signatures. The more signatures there have already been collected and the more signatures there are already in the sample set, the less likely it will be that the collector is allowed to add another collected signature to this sample set.

We have to agree in advance on the selection process for signatures that are allowed to get into the sample. This selection process will not be under the full control of the collector. It is designed such that the sample size grows logarithmically with the number of collected signatures. The size of the sample becomes this way a probabilistic indicator for the logarithm of the number v of collected signatures. An inspection of the $O(\log v)$ sample signatures alone allows us then to verify that the sampling process was performed properly and that at least around v signatures must have been given to the collector.

We assume that the following public-key infrastructure is in place:

Every person A who is entitled to participate in the planned type of signature collection is in possession of a private signing key K_A and a public verification key K'_A for some deterministic digital signature scheme. Each such person A has also received a certificate C_A that confirms for K'_A that the corresponding K_A is the only secret key that is available for A to be used in the collection. It is the responsibility of the certification authority that generated C_A to ensure that it is very difficult for anyone to obtain simultaneously valid certificates for more than

one single signing key and that it is very difficult for not-entitled persons to get such a certificate.

The signature scheme under which the K_A and K'_A are used must be deterministic. By this we mean that for each message M, person A can only generate one single signature value that verifies under K'_A. The signature must not contain any freely selectable entropy. For instance using RSA signatures in the form of the `RSASSA-PKCS1-v1_5` scheme [2] and a fixed message digest function (say SHA-1) would be suitable. Using for example RSA with the PSS scheme [3] or DSS [4] to generate the signature, or leaving the signer a choice of multiple algorithms, must not be allowed. Otherwise a signing key owner could generate many different valid signatures for the same message M, which could allow her to get her signatures counted several times in the collection.

The signature collection is performed as follows:

1. The signature collector distributes to all potentially interested signers A the proposed message M together with an indication of which types of public-key certificates are accepted in this collection.

2. If person A decides not to sign M, she either does nothing or just confirms that she has received the proposal and is not interested in supporting it.

3. If A supports the proposal M, she generates the signature $s_A = \text{sign}(K_A, h(M))$ by applying her signing key K_A on a message digest of M, as required by the signature scheme. The resulting signature s_A is submitted together with the verification key K'_A and the certificate C_A to the signature collector.

4. The collector has a storage space consisting of n slots, each of which can save one signature, as well as the corresponding public key and certificate (or a pointer to it in some directory). The collector knows the verification keys for all certificate types that are allowed to be used.

5. The collector determines for each newly received signature the *slot position*

$$t = \varphi(H(s_A)),$$

where H is a uniformly distributed secure hash function that maps signatures onto binary words in the range 0 to $2^l - 1$, and

$$\varphi : \{0, \ldots, 2^l - 1\} \to \{1, \ldots, n\}$$

is a function that maps a uniformly distributed hash value onto one out of n non-uniformly distributed slot numbers (typical practical values are $l = 64$ and $100 < n < 5000$). A precise construction for φ will be suggested below.

6. If the collector has already stored a signature in slot number t, then the newly received signature will simply be discarded (see also section 4.4 for a better approach). Otherwise, the collector verifies that the received certificate C_A and the verification key K'_A are valid and that s_A is valid under K'_A for message M. If so, he will store s_A, K'_A, and C_A in slot number t.

7. Many collectors can be active in the same signature collection at the same time. The merged collection result will contain one sample signature for each slot for which at least one of the collectors has received a signature. If the same signature is submitted multiple times or to different collectors, this will not influence the number of filled slots. The same signature will always fall into the same slot with each collector and will therefore appear at most once in the end result.

It is intuitively clear that to find a signature whose hash has twenty leading zeros, we need to look at about 2^{20}, or a million, signatures. In what follows, we show how to count more precisely based on this general idea.

Let u be the number of slots that the collector has been able to fill with verifiable signatures. This number will be used to estimate the number v of valid distinct signatures that have been collected. The number u can be verified quickly by anyone who has reliable access to the public verification keys of the certification authorities, because for every filled slot number, one sample signature and the associated certificate has been kept as a proof that at least one signature for this slot was collected.

We describe the number of collected signatures using the discrete random variable V. The discrete random variables U_i with $1 \le i \le n$ describe the number of filled slots with slot numbers in the range 1 to i and $U = U_n$ shall be the random variable describing the total number of signatures that the collector has stored. The dependency between U and V is determined by the function φ, which defines the probability distribution of the random variable T that assigns signatures to slot numbers.

Let $p_t = P(T = t)$ be the probability that a uniformly distributed l-bit input hash value is mapped

by φ onto slot number t, that is

$$p_t = \frac{\left|\{w \mid 0 \le w < 2^l \wedge \varphi(w) = t\}\right|}{2^l}, \quad \sum_{t=1}^{n} p_t = 1.$$

Given a distribution of T in the form of p_t values, a suitable φ can be constructed as

$$\varphi(w) = \min\left\{t \mid 1 \le t \le n \wedge 2^l \cdot \sum_{i=1}^{t} p_i > w\right\}$$

and easily implemented as a binary search in a stored table of the values

$$w_i = \left\lceil 2^l \cdot \sum_{t=1}^{i} p_t \right\rceil, \text{ for all } i \in \{1, \ldots, n-1\}.$$

The word size l of the secure hash function H has to be selected sufficiently large such that $2^{-l} \ll p_t$ for all $1 \le t \le n$.

After v different valid signatures have been collected, the probability that slot t is still empty will be $(1-p_t)^v$, which we will abbreviate in the following as $q_t := (1-p_t)^v$.

The probability that after v different signatures have been collected, exactly u signatures are stored in the first $1 \le i \le n$ slots is

$$r_i(u,v) := P(U_i = u | V = v) = \sum_{\substack{F \subseteq \{1,\ldots,i\} \\ |F|=u}} \left(\prod_{t \in F}(1-q_t)\right)\left(\prod_{t \in \{1,\ldots,i\} \setminus F} q_t\right).$$

The above function can in spite of its exponentially long sum be computed efficiently in $O(n^2)$ time using the following recursion over the number of slots:

$$r_i(u,v) = \begin{cases} r_{i-1}(u,v) \cdot q_i + \\ r_{i-1}(u-1,v) \cdot (1-q_i), \\ \qquad \text{if } 0 \le u \le i \text{ and } i > 0 \\ 1, \quad \text{if } i = u = 0 \\ 0, \quad \text{otherwise} \end{cases}$$

We can now determine the expected number of filled slots among the first i slots recursively as

$$\begin{aligned} \overline{u_i} &:= E[U_i | V = v] = \sum_{u=0}^{i} u \cdot r_i(u,v) \\ &= \sum_{u=0}^{i} u \cdot [r_{i-1}(u,v) \cdot q_i + \\ &\quad r_{i-1}(u-1,v) \cdot (1-q_i)] \\ &= q_i \left(\sum_{u=0}^{i-1} u \cdot r_{i-1}(u,v) + i \cdot r_{i-1}(i,v)\right) + \\ &\quad (1-q_i) \sum_{u=-1}^{i-1} (u+1) \cdot r_{i-1}(u,v) \\ &= q_i \cdot (\overline{u_{i-1}} + 0) + (1-q_i) \cdot \\ &\quad \left(\sum_{u=0}^{i-1} u \cdot r_{i-1}(u,v) + \sum_{u=0}^{i-1} r_{i-1}(u,v)\right) \\ &= q_i \cdot \overline{u_{i-1}} + (1-q_i) \cdot (\overline{u_{i-1}} + 1) \\ &= \overline{u_{i-1}} + 1 - q_i, \end{aligned}$$

and therefore

$$\overline{u_i} = E[U_i | V = v] = \sum_{t=1}^{i} (1 - (1-p_t)^v).$$

We can similarly calculate the average square of the number of filled slots among the first i slots recursively as

$$\begin{aligned} \overline{u_i^2} &:= E[U_i^2 | V = v] = \sum_{u=0}^{i} u^2 \cdot r_i(u,v) \\ &= q_i \left(\sum_{u=0}^{i-1} u^2 \cdot r_{i-1}(u,v) + i^2 \cdot r_{i-1}(i,v)\right) + \\ &\quad (1-q_i) \sum_{u=-1}^{i-1} (u+1)^2 \cdot r_{i-1}(u,v) \\ &= q_i \cdot (\overline{u_{i-1}^2} + 0) + \\ &\quad (1-q_i) \cdot (\overline{u_{i-1}^2} + 2\overline{u_{i-1}} + 1) \\ &= \overline{u_{i-1}^2} + 2\overline{u_{i-1}} + 1 - q_i \cdot (2\overline{u_{i-1}} + 1) \end{aligned}$$

and therefore

$$\overline{u_i^2} = E[U_i^2 | V = v] = \sum_{t=1}^{i} (2\overline{u_{t-1}} + 1) \cdot (1 - (1-p_t)^v).$$

This way, we can determine the variance of the number of filled slots as

$$\begin{aligned} \mathrm{Var}[U|V=v] &= \sigma^2 = E[(U - E[U])^2 | V = v] \\ &= E[U^2 | V = v] - E^2[U | V = v] = \overline{u_n^2} - \overline{u_n}^2. \end{aligned}$$

3.1 Selecting Slot Probabilities

Ideally, we should aim to choose the slot selection probabilities p_t and the corresponding φ in a way such that we get

$$E[U | V = v] \approx a \ln(v+1)$$

for some scaling factor a. Then $e^{\frac{u}{a}} - 1$ would be the corresponding estimated number of collected unique

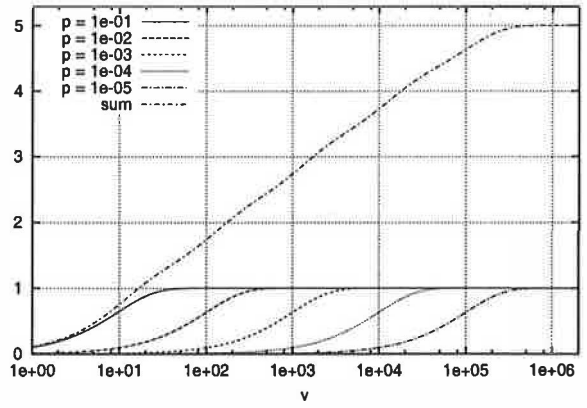


Figure 1: In this graph, we can see the function $1-(1-p)^v$ with the five parameter values $p = 10^{-1}, 10^{-2}, \ldots, 10^{-5}$ on a logarithmic scale for v, as well as the sum of these five curves. The $1-(1-p)^v$ graphs form smoothed unit steps near $1/p$ and the sum approximates a straight line (i.e., a logarithm) in the range 10^1 to 10^5.

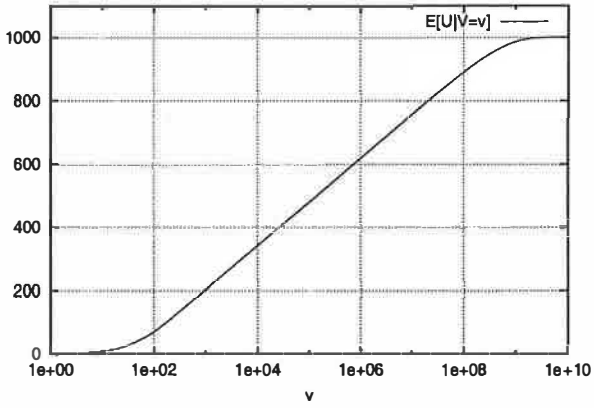


Figure 2: This graph shows the expected number of filled slots $E[U|V=v]$ over the number of collected signatures v for geometrically distributed slot selection probabilities p_t with parameters $n = 1000$ and $\beta = 0.9835$, selected such that up to $\hat{v} = 10^9$ signatures can be counted without danger of saturating the scheme.

valid signatures. The factor a determines the size of the sample set, and therefore the resolution and relative error of the estimated size of the collection. A logarithmic mapping between u and v corresponds to a constant relative error of the estimate of v over the entire range.

A good practical choice for φ turns out to be a geometric distribution of the form

$$p_t = \alpha\beta^t \tag{1}$$

with

$$\alpha = \left(\sum_{t=1}^{n}\beta^t\right)^{-1} = \frac{1-\beta}{\beta-\beta^{n+1}} \quad \text{and} \quad 0<\beta<1.$$

The reason that a geometric distribution leads to the desired result becomes intuitively clear as follows. We look at the graph of the function $1-(1-p)^v$ as we vary parameter p using values of a geometric series such as $p = 10^{-1}, 10^{-2}, \ldots, 10^{-5}$. These graphs are shown in Fig. 1, where v is plotted on a logarithmic scale. We see that this term results in a smooth unit-step function, and the location of the step coincides roughly with $1/p$, because if we solve $1-(1-p)^v = 1-e^{-1} \approx 0.63$ for v, we get

$$v = \frac{-1}{\ln(1-p)} \approx \frac{1}{p} \qquad \text{for } 0 < p \ll 1.$$

Since $E[U|V=v]$ is a sum of terms $1-(1-p)^v$, we can approximate on a logarithmic scale for v a straight line using equidistant values for $1/p$. This is demonstrated by the graph of the sum of the five smooth unit-step functions that is also shown in Fig. 1. It forms a nearly straight line in the range 10^1 to 10^5. This way, a geometric series of parameters p_t has resulted in $E[U|V=v]$ becoming an approximation for a logarithm of v.

Figure 2 shows $E[U|V=v]$ with a more practical set of parameters. In this example we use $n = 1000$ slots and we want to be able to handle up to $\hat{v} = 10^9$ signatures, for which $\beta = 0.9835$ is a good choice. With these parameters, the average relative error will be around 10% and therefore collection sizes that are less than a factor of two apart can still be separated very reliably. Using this parameter set, Fig. 3 shows for a number of collection sizes v the distribution $P(U=u|V=v)$. If a higher resolution is desired, the number n of slots has to be increased and β has to be adjusted accordingly.

If the desired number of slots n and the maximum expected signature count $\hat{v}$ are given, a suitable value for β fulfills the equation

$$1-(1-p_n)^{\hat{v}} = 1-e^{-1} \approx 0.63$$

such that the last slot n will be filled with a probability of a bit over one half after $\hat{v}$ signatures have been collected. The equation can be solved for β numerically using a binary search. Figure 4 shows for the practically useful ranges of n and $\hat{v}$ the corresponding β values and can be used to select a suitable β graphically.

With the geometric series of slot probabilities from (1), the probability densities $P(U=u|V=v)$

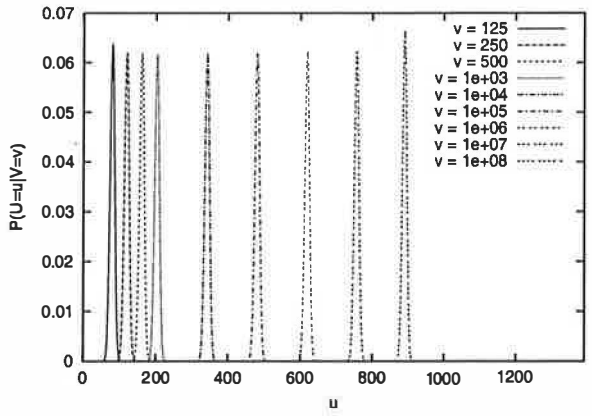


Figure 3: In this graph, the probability density $P(U = u|V = v)$ is shown for a number of signature collection sizes v. The parameters n and β are the same as in Fig. 2.

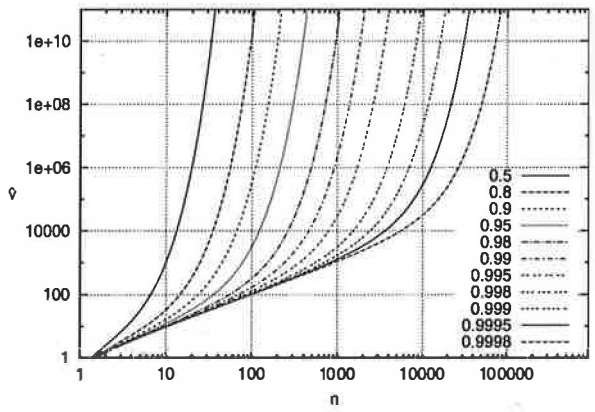


Figure 4: Using this graph, a suitable base β can be selected for a given number of slots n and a maximum expected signature count $\hat{v}$.

can be approximated by a Gaussian normal distribution:

$$P(U = u|V = v) \approx \frac{e^{-\frac{(u-\overline{u_n})^2}{2\overline{u_n^2}-2\overline{u_n}^2}}}{\sqrt{2\pi\left(\overline{u_n^2}-\overline{u_n}^2\right)}}$$

This works well at least as long as the variance is larger than one and the expected value is at least several standard deviations away from 0 and n, otherwise boundary effects make the shape of the distribution noticeably asymmetric.

3.2 Estimating the Collection Result

We have seen how we can efficiently determine the distribution $P(U = u|V = v)$, but in order to interpret the given outcome u of a probabilistic signature

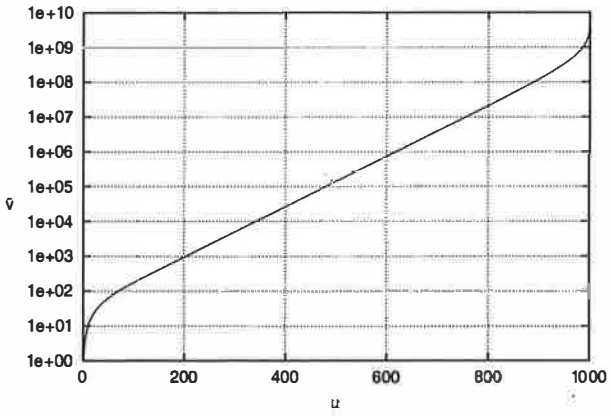


Figure 5: For the same parameters n and β as in Fig. 2, this graph shows the function $\tilde{v}$ that provides the maximum-likelihood estimate of v for a given u.

collection, we are more interested in the Bayesian distribution

$$P(V = v|U = u) = \frac{P(U = u|V = v) \cdot P(V = v)}{\sum_{v'} P(U = u|V = v') \cdot P(V = v')}$$

that tells us how probable every possible number of signatures v was, given the outcome u. Unfortunately, we cannot practically determine this quantity unless we have a reasonable estimate for the distribution of V.

Therefore, the best practical estimate for v given a certain u will be the maximum-likelihood choice, i.e. the v for which $P(U = u|V = v)$ is maximal. Given the p_t values, it is possible to precalculate a function

$$\tilde{v} : \{0, \ldots, n\} \to \mathbb{N}$$

with $\tilde{v}(0) = 0$ and

$$P(U = u|V = \tilde{v}(u)) \geq P(U = u|V = v)$$

for all $u, v \in \mathbb{N}$ with $0 \leq u < n$, $0 \leq v$ and $P(U = n - 1|V = v) > P(U = n|V = v)$. The value $\tilde{v}(n)$ remains undefined, because if all slots are filled, the only estimate about v that we can make is that $v > \tilde{v}(n-1)$, a case which should be avoided in applications by selecting a generously large value of $\hat{v}$.

For practical purposes, it is usually sufficient to determine $P(U = u|V = v)$ for v increasing in steps of around 5% and then tabulate in $\tilde{v}(u)$ that v that resulted in the largest $P(U = u|V = v)$. Figure 5 shows the result for our example parameter selection.

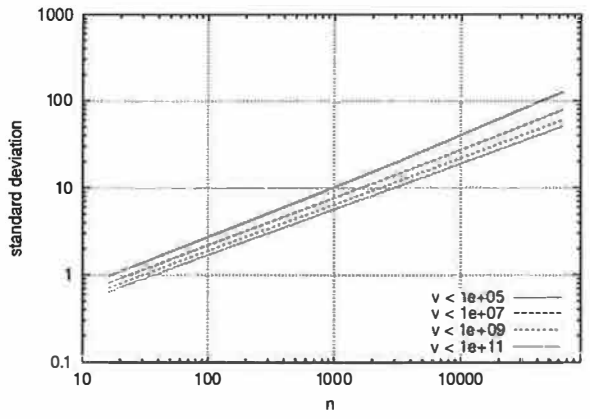


Figure 6: This graph shows for various n and $\hat{v}$ values the standard deviation $E[(U - E[U])^2]^{\frac{1}{2}}$ when $V = \tilde{v}(\frac{n}{2})$. It grows proportional to $\sqrt{n}$.

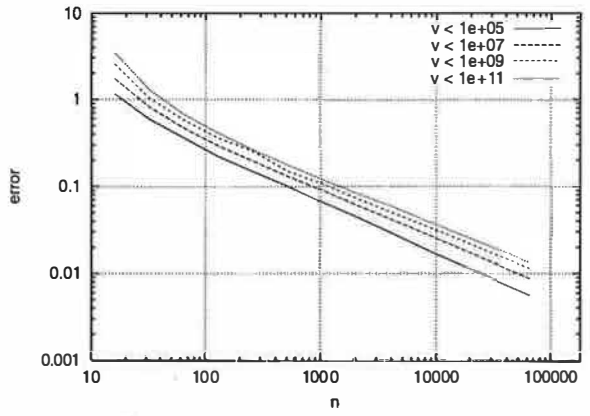


Figure 7: The root-mean-square of the relative error $(E[(\tilde{v}(U) - V)^2/V^2]^{\frac{1}{2}}$ for $V = \tilde{v}(\frac{n}{2}))$, which this graph shows for various n and $\hat{v}$ values, falls proportional to $1/\sqrt{n}$.

Alternatively, $\tilde{v}$ can also be determined simply by inverting $E(U|V = v)$, which is practically equivalent since the expected value happens to coincide well with the peak of $P(U = u|V = v)$ (see Fig. 5 compared to Fig. 2).

Figure 6 shows how the standard deviation of U depends for a fixed v on the parameters n and $\hat{v}$. It depends only slightly on $\hat{v}$ and grows proportional to $\sqrt{n}$. The root-mean-square of the relative error of the estimate $\tilde{v}$ develops accordingly proportional to $1/\sqrt{n}$ (Fig. 7).

Instead of just looking at the number u of collected signatures, the verifiers can also base their maximum-likelihood estimate of v on the set $F \subseteq \{1, \ldots, n\}$ of slot positions that have been filled. They can then make their estimate using the function $\tilde{v}(F)$ that returns that value of v which maximizes

$$\prod_{t \in F} (1 - (1 - p_t)^v) \cdot \prod_{t \notin F} (1 - p_t)^v.$$

This approach allows the verifiers to utilize all available information, but it also makes it impractical to precompute the function $\tilde{v}$ as a simple table.

Verifiers do not necessarily have to check the content of all slots. If many filled slots with low p_t have been verified, all those slots with orders-of-magnitude larger p_t will most likely be full as well. This allows the result to be represented even more compactly. The performance of the scheme could also be improved by using several slot mapping functions φ of different granularity. When a collector's storage space becomes full, he switches to the next φ and merges existing slots accordingly.

4 Security Considerations

4.1 Trust Placed in Collectors

Collectors can easily discard signatures and can therefore make the number of submitted signatures look smaller than it actually was. As a consequence, the result of a signature collection can only be meaningful if signatures were collected by someone with a genuine incentive to collect as many signatures as possible.

It is also worth noting that participating signers have no guarantee that signature collectors store only a maximum of n signatures. While signature collectors are only required to store and present $u = O(\log v)$ signatures to the verifiers, nothing prevents them from creating a record of all v signatures. The privacy protection that signature collection offers thus protects only the signer from the verifier, but not the signer from the signature collector.

4.2 Bribery

If either collectors or the signing key holders know in advance which signing keys are able to fill a slot, this might with some applications lead to the creation of a market for those sample signatures that fill slot positions with low p_t values. The resulting risk is that instead of convincing $O(v)$ people of the value of proposition M, the collectors find it easier to secretly purchase suitable sample signatures from $O(\log v)$ selected signers to achieve the same result.

The presented scheme assigns the slot number $t = \varphi(H(s_A))$ based on participant A's signature $s_A = \text{sign}(K_A, h(M))$ of document M. It is the

very nature of a digital signature that its value is not predictable by anyone except the holder of the signing key. This way, the signature collectors have no way of identifying in advance those signing key holders who could provide them with the valuable sample signatures that will fall into slots with low p_t and that therefore represent a large number of signatures. If the slot number were instead assigned based on the signer's identity and not her signature of M, then the signature collector could identify in advance those few signers whose signatures they need to fill the low-probability slots. Strong incentives could be offered to these few individuals to sign the proposal.

If φ and H are publicly known, then all potential signers can check whether their signature would fall into a slot with a low p_t value. They could contact the signature collectors secretly and could offer to sell their more valuable signatures.

In applications where this second kind of bribery is of concern, a secret nonce N only known to the collectors should be concatenated with the signature in the calculation of the slot position

$$t = \varphi(H(N \parallel s_A)).$$

This will prevent signers from determining whether their signature could fill an unlikely slot. The nonce N will be published *after* the signature collection process has finished, as this parameter has to be accessible to the verifiers. Every sample signature that is stored in a slot position has to be registered at a timestamping service *before* the publication of N. The early leakage of N into the public only becomes a problem if the number of signers who hear about N during the collection is of an order of magnitude comparable to the number of signatures that the collectors will later claim to have obtained. In this case, it is also very likely that someone opposing the proposal will learn N *before* the collection closes. This person can later prove the information leakage by registering N immediately at a time stamping service, and thus gains strong evidence for early public knowledge of N, which can then invalidate the entire signature collection.

4.3 Collusion of Signers

Another threat that has to be taken into account is that a group of signing key holders could collude with the signature collectors before the start of the collection. They could generate a very large number of candidate documents M with slight variations until they find one M for which the signing keys of the members of that group can be used to fill an unusually high number of slots. They then start a signature collection campaign with this carefully phrased proposal M that guarantees their own secret keys a high impact.

If c signers collude with the signature collectors, then the probability that an arbitrary message M will allow these c signers to fill at least w slots is

$$\sum_{i=w}^{n} P(U = i|V = c).$$

In order to find with sufficient probability (say $1 - e^{-1} \approx 63\%$) such a message M that fills at least w slots, they would have to generate at least

$$\begin{aligned} m &= \frac{-1}{\ln\left(1 - \sum_{i=w}^{n} P(U = i|V = c)\right)} \\ &\approx \frac{1}{\sum_{i=w}^{n} P(U = i|V = c)} \end{aligned}$$

candidate messages M and then have to calculate all cm signatures on these to determine the slot position of each signing key and message pair. This should become computationally very impractical when $cm \gg 10^{20}$.

Figure 8 shows the example distributions from Fig. 3 on a logarithmic scale. Like a Gaussian normal distribution, the $P(U = u|V = v)$ curves have a parabolic shape on a logarithmic scale and fall off rapidly. It is unlikely that a group of c signers who collude with the signature collectors can increase the number w of illegitimate sample signatures to more than nine standard deviations above $E[U|V = c]$.

In addition, if the true number of collected signatures v is larger than $\tilde{v}(w)$, then the collusion is unlikely to have any significant effect on the final $\tilde{v}(u)$, because most of the w slots filled by keys of the colluding group would have been filled anyway by legitimate signatures from non-colluding signers, and then the w illegitimate sample signatures do not contribute any more to the estimate $\tilde{v}(u)$.

A collusion could be prevented by requiring the signature collectors to announce M to some trusted party, which in return generates a nonce that has to be concatenated with M before the entire signature collection can start. This way, the signature collectors cannot carefully select M to maximize the impact of the colluding signers. For such a mechanism to be effective, however, the rate at which such announcements of M are allowed has to be limited, as otherwise, the announcement would not prevent the generation of a large number of candidate messages. Such a rate limitation, however, would defy the design goal of keeping the signature collection scheme free of central facilities and bottlenecks.

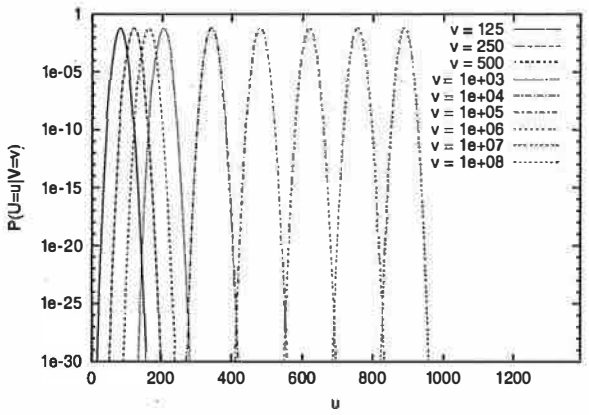


Figure 8: This graph is identical to Fig. 3 but shows the probabilities on a logarithmic scale.

4.4 Fair Sampling

An indication for a collusion of signers would be if an unusually large number of sample signatures came from signers who are likely to be affiliated with the collectors. This effect can also appear if only the first received signature for each slot position becomes the sample signature for that slot. It would not be unusual for signers in the social context of the collectors to submit their signatures earlier than most others and thus have a higher probability of ending up as sample signers.

For this reason—as well as in the interest of general fairness—it is advisable for the signature collectors to give every submitted signature an equal chance of becoming a sample. This can be achieved by keeping a counter g_t for every slot t, which indicates how many valid signatures have already been received for this slot. Whenever a new signature is received for slot t and is validated successfully, then g_t is increased by one, and the newly arrived signature will replace the previously stored one with probability $1/g_t$. This ensures that within a slot position, all submitted signatures have an equal probability of becoming the final sample, independent of the order in which they arrive.

This technique works because, if $k_i = 1/i$ is the probability for keeping the i-th arriving signature and n signatures are received in total, then the probability f_i with which the i-th signature becomes the final surviving sample is

$$f_i = k_i \cdot \prod_{j=i+1}^{n} (1 - k_j) = \\ = \frac{1}{i} \cdot \frac{i}{i+1} \cdot \frac{i+1}{i+2} \cdots \frac{n-1}{n} = \frac{1}{n}$$

and therefore equal for all arrival positions i.

If signature collection results from the collectors $L_1, \ldots, L_d$ are merged, then every collector L_i provides for each slot t his submission count $g_{t,i}$. With probability $g_{t,i} / \sum_{j=1}^{d} g_{t,j}$ the sample signature in slot t of L_i will go into slot t of the merged result, and the submission count of this slot will be set to $\sum_{j=1}^{d} g_{t,j}$. This maintains the equal chance for all signature submissions to end up in the final result, provided that signers cooperate and do not submit signatures multiple times. Partial collection results should be merged only in a hierarchical way.

4.5 Cryptographic Assumptions

Signing messages that have been generated by others requires some diligence to avoid exposure to various attacks. The chosen signature scheme should certainly be robust against adaptive chosen message attacks. In addition, the syntax of the proposed message M should be sufficiently restricted to avoid that M can have any other uses then being interpreted as a proposal that was written for a signature collection.

It has been suggested to avoid signing messages without adding some entropy first [3], but the addition of random "salt" values seems not to be an applicable option against hypothetical future attacks in this application. The digital signature design community is therefore invited to consider signature collection based on probabilistic counting—apart from the well known subliminal-channel concerns of salt values—as yet another good reason for not giving up deterministic signature algorithms and to continue their design, evaluation, and standardization.

It is important to note that the presented scheme relies on a property of the utilized deterministic digital signature method that is not among those commonly studied in-depth. We assume that it is infeasible with the used signature scheme to generate an *ambiguous verification key* for which there is a practical way to generate many different signatures for the same document M that all verify correctly under this single public key[1]. An otherwise perfectly safe deterministic signature scheme might be unsuitable or at least require additional tests by the certification authority or the verifier and signature collectors on the public keys in order to keep individual signers from submitting many different seemingly valid signatures. If the chosen signature scheme allows the generation of ambiguous verification keys that

[1] With RSA, there exists exactly one signature s for $h(M)$ under public key (e, n) (with $s^e \bmod n = h(M)$), as long as $\gcd(e, \phi(n)) = 1$.

cannot be detected later, then the certification authority will have to generate the key pair and certify for the verification key not only the identity of the owner, but also that this key verifies only one single signature for any document.

5 Related Work

The general idea of *probabilistic counting* was probably first introduced by Morris [5], who used a small integer register to estimate how often an event had occurred. The number of events can be considerably larger than the maximum value that the register can hold, therefore the register is increased by one only with a certain probability when a new event occurs, and this probability is reduced exponentially as the register value increases. The register value becomes an estimate for the logarithm of the number of events. The logarithmic relationship keeps the relative error constant.

The probabilistic counting concept was later extended by Flajolet and Martin [6] to estimate the number of different values in a large database table. In their algorithm, every value is transformed using a hash function into a word. The position of the first 1-digit in this word is determined, and the corresponding bit is set in a bitmap. The position of the first zero in the resulting bitmap is used as an estimate for the logarithm of the number of different values that were processed this way. Multiple identical values are automatically discarded, as they would only set the same bit several times. The counting process can easily be distributed and the resulting bitmaps can be combined using a logical-*or* operation. The probability that the first 1-digit in a uniformly distributed word is the n-th digit is 2^{-n}, and the position of the first zero digit in the bitmap is therefore comparable to the value of Morris' register. This technique was later extended by Kirschenhofer, Prodinger and Szpankowski by using a histogram with saturation arithmetic instead of a bitmap to increase counting accuracy [7].

6 Conclusions

The cryptographic equivalent of signature collection can be implemented using a probabilistic counting technique that provides efficient verification of the result. We have developed and discussed a practical construction of such a process with application properties very similar to those of handwritten signature collections. The presented numerical analysis will help in the selection of suitable parameters. We have discussed how Web page metering, TV rating and opinion gathering, newsgroup contribution ranking, and joint administration concepts are among the possible application areas. This signature collection technique can lead to improvements in robustness, privacy protection, and efficiency, as long as the application designer fully understands the described security considerations. Perhaps this application idea will encourage the development of public-key infrastructures which support a new key and certificate type that provides the assurance that an individual cannot easily generate more than one verifiable signature for a message.

The author wants to thank Ross Anderson, David Wheeler and the referees for valuable comments.

References

[1] Moni Naor, Benny Pinkas: Secure and Efficient Metering. In Kaisa Nyberg (ed.): *Advances in Cryptology – EUROCRYPT '98*, LNCS 1403, pp. 576–590. Springer-Verlag, May/June 1998.

[2] Burt Kaliski, Jessica Staddon: PKCS #1: RSA Cryptography Standard – Version 2.0. RSA Laboratories, September 1998. `http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/`

[3] Mihir Bellare, Phillip Rogaway: The Exact Security of Digital Signatures – How to Sign with RSA and Rabin. In Ueli Maurer (ed.): *Advances in Cryptology – EUROCRYPT '96*, LNCS 1070, pp. 399–416. Springer-Verlag, May 1998.

[4] Digital Signature Standard (DSS). Federal Information Processing Standards Publication FIPS PUB 186-1, U.S. Department of Commerce, December 1988.

[5] Robert Morris: Counting Large Numbers of Events in Small Registers. *Communications of the ACM*, Vol. 21, No. 10, pp. 840–842, October 1978.

[6] Philippe Flajolet, G. Nigel Martin: Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, Vol. 31, No. 2, pp. 182–209, October 1985.

[7] Peter Kirschenhofer, Helmut Prodinger, Wojciech Szpankowski: How to Count Quickly and Accurately: A Unified Analysis of Probabilistic Counting and Other Related Problems. In W. Kuich (ed.): *Automata, Languages and Programming*, 19th International Colloquium, pp. 211–222, Wien, Austria, 13–17 July 1992. Springer-Verlag.

Can Pseudonymity Really Guarantee Privacy?

Josyula R. Rao Pankaj Rohatgi
I.B.M. T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
{jrrao, rohatgi}@watson.ibm.com

Abstract

One of the core challenges facing the Internet today is the problem of ensuring privacy for its users. It is believed that mechanisms such as anonymity and pseudonymity are essential building blocks in formulating solutions to address these challenges and considerable effort has been devoted towards realizing these primitives in practice. The focus of this effort, however, has mostly been on hiding explicit identity information (such as source addresses) by employing a combination of anonymizing proxies, cryptographic techniques to distribute trust among them and traffic shaping techniques to defeat traffic analysis. We claim that such approaches ignore a significant amount of identifying information about the source that leaks from the contents of web traffic itself. In this paper, we demonstrate the significance and value of such information by showing how techniques from linguistics and stylometry can use this information to compromise pseudonymity in several important settings. We discuss the severity of this problem and suggest possible countermeasures.

1 Introduction

1.1 The Problem

Privacy in the computer age is the right of individuals to protect their ability to selectively reveal information about themselves so as to negotiate social relationships most advantageous to them. The complete absence of privacy protection in the basic Web infrastructure coupled with the increased migration of human activity from real–word interactions to web based interactions poses a grave threat to the privacy of the entire human population. Increasingly sophisticated databases of profiles of individuals based on their web interactions are being built and exploited. Clearly, assuring privacy has become one of the fundamental challenges facing the Internet today.

1.2 Background

A number of tools and techniques have been proposed and deployed to address the privacy concerns of Internet users. To place these in perspective, it is important to understand two primitives, *anonymity* and *pseudonymity*, that are considered to be the core building blocks of most privacy solutions.

Anonymity refers to the ability of an individual to perform a *single* interaction with another entity (or set of entities), without leaking any information about his/her identity. While anonymity is very effective at protecting the identity of an individual, its ephemeral nature makes it ill–suited for most kinds of web interactions.

This shortcoming is addressed by the concept of pseudonymity, which enables an individual to participate in a series of web interactions, all linkable to a single identifier (also known as a *pseudonym*), with the guarantee that the pseudonym cannot be linked back to the individual's identity. The persistence of pseudonyms permits the establishment of long term web–relationships. The ability of an individual to choose different pseudonyms for different activities enables an individual to further protect his/her privacy by partitioning his/her interactions into unlinkable activities.

Broadly speaking, technologies for realizing privacy fall into four different categories [5].

- Anonymizing agents ensure that packets from a user cannot be linked to his/her IP address. A simple anonymizing agent is just a single trusted third party (e.g. the Anonymizer [1]) that decrypts and redirects requests from users. Typically, the third party disassociates the initiator's identity from the recipient's identity by replacing the user identifying headers with its own headers. This approach is deficient in that it presents a single point of failure and is vulnerable to traffic analysis. Therefore, more sophisticated schemes use cryptographic techniques to distribute trust in a *chain* of third parties and employ traffic shaping techniques to defeat traffic analysis. This ensures that anonymity/pseudonymity is not compromised even in the presence of a few compromised third parties.

  Examples of such technologies include Anonymizer[1], Crowds [17], Onion Routing [8] and the IP Wormhole technology used in the Freedom network. [9].

- Application level filters to eliminate obvious identity information from an individual's web traffic, such as name, e–mail address and affiliation etc. Examples include the Freedom word scanner [10].

- Pseudonym agents which manage pseudonyms to develop persistent relationships. Examples include Lucent Personalized Web Assistant (LPWA) [7], P3P [16] and Freedom Network [9].

- Negotiation Agents/Trust Engines which negotiate on a user's behalf and determine when a user's privacy policies are satisfied. Examples include software tools for cookie management and P3P [16].

In practice, a service providing privacy would incorporate elements of some or all of these technologies. It has been the general belief that appropriate use of these tools and technologies should be adequate to counter threats to privacy, provided that the cryptography used is sufficiently strong, sufficient numbers of trusted anonymizing parties remain uncompromised and adequate traffic analysis countermeasures are in place. In fact, considerable energy has been devoted to perfecting and commercializing these technologies.

1.3 Contributions

The goal of this paper is to argue that the current focus on hiding explicit identity information is not sufficient to guarantee privacy. There is still a considerable wealth of identifying information in the body of the communication messages, which to the best of our knowledge, is only minimally altered by the existing tools and privacy infrastructure, before it is delivered to the recipient. At best, what may be removed is the obvious identity information such as header blocks, signature blocks, addresses, names, etc. However, the body of each communication leaks identifying information in many subtle ways. Although these leaks may be subtle, with a sufficient amount of communication, it becomes feasible to distill useful identifying information. This is especially true for pseudonymous communications where the recipient is a newsgroup, a chat–room, a mailing list or any server that stores the communication in an accessible manner. This information is usually publically archived and available for analysis for all eternity.

We can classify the information available in such pseudonymous communications into two categories:

- *Syntactic*: This term is intended to encompass information dealing with the components of the message and their structural layout. This could include information such as vocabulary, sizes of sentences and paragraphs, use of punctuation, frequency of blank lines, common misspellings, etc.

- *Semantic*: This term is intended to encompass the concepts used in the communication and the language used to express these concepts.

There is field of linguistics, known as stylometry, which uses both these categories of information to ascribe identity or authorship to literary works. This field has had remarkable success in authenticating the authorship of such diverse works such as the disputed *Federalist* papers [14, 15, 2], the Junius Letters [6], Greek prose [13], Shakespeare's works[3, 12, 18]. More recently, stylometry was used by Prof. Donald Foster at Vassar College to attribute the authorship of the novel "Primary Colors" to Joe Klein, an attribution subsequently confirmed by handwriting analysis and the author's own admission.

We believe (and demonstrate) that recent advances in stylometry pose a significant threat to privacy that merits the serious and immediate attention of the privacy community. For instance, using stylometry, one can link the multiple pseudonyms of a person and if one such pseudonym happens to be his/her identity, then the protection afforded by the other pseudonyms is compromised.

For instance, given a pseudonym for an identity and a moderately sized list (say around one hundred entries) of other pseudonyms, at least one of which corresponds to the same identity, we show that it is easy to perform semantic stylometric tests to correctly link the pseudonyms with good probability, provided enough data (a few thousand words per pseudonym) is available. In cases where, the list is very long, perhaps with a million or more entries, the same technique can be adapted to cut down the possibilities by a large factor.

In fact, the two parameters of the technique are (a) the data requirements per pseudonym and (b) the discriminating power of the technique, that is, the factor by which the number of possibilities is reduced by the technique. We believe that the discriminating power improves with increased data requirements. Our experience with this technique, leads us to conclude that the values of these parameters are such that the privacy of users of anonymous short e-mail messages or short newsgroup postings is not yet at risk but authors (anonymous or pseudonymous) of at least few thousand words are susceptible to our techniques and analysis.

Note that these results and estimates are based on a technique borrowed from stylometry that does not make any use of the wealth of other types of identifying information (such as misspellings, structural layout, etc) that is present in most types of web traffic. This is because, linguists have focussed mostly on proof-read and typeset documents such as books and journals. We believe that approaches that also incorporate this additional information could be even more effective.

While it may seem that the leakage of identifying information from the syntax of messages can be easily remedied by an application specific agent run on the user's behalf, the problem with the leakage from semantics is much more insidious and difficult to eliminate satisfactorily. Particularly egregious is the fact that this line of attack cannot be fixed without changing the meaning of the message. Perhaps with collaboration and close cooperation between linguistics, statisticians and the privacy community, it may be possible to develop a much more intelligent and usable agent to minimize semantic leakage without degrading the message quality and meaning.

1.4 Plan of Paper

In section 2, we begin describing in detail the types of identifying information available in the specific setting of newsgroup postings. In the following section, we describe a specific stylometric technique, called principal component analysis of function words which can be applied for attacking pseudonymity. In section 4, we describe our experiments with this technique on data collected from newsgroup archives and mailing lists and the results obtained. In section 5, we discuss several other promising avenues to further improve the basic attack by incorporating other types of leaked information. We conclude by presenting a discussion on possible countermeasures.

2 Exposures in Newsgroup Postings

Newsgroup postings are frequently archived and are freely available over the Internet. Typically, each valid (RFC-1036 compliant) posting, consists of "RFC-822 style" headers (including the *From:*, *To:*, and *Subject:* fields), a body and (optionally) a signature. Pseudonymous postings would at best hide all identifying information (except the pseudonym) in the header fields and would be unlikely to contain any obvious identifying information in the signature or the body.

As mentioned earlier, in spite of identity hiding techniques, there still are two types of identifying information in the body. The first category, termed as *syntactic information* includes:

- *Length of sentences and paragraphs.*
- *Number of words and paragraphs.*
- *Paragraph style:* This includes the indentation style and the spacing between paragraphs.
- *Words, Phrases and their frequency:* This includes the vocabulary of the user.

- *Misspelled words and their frequency.*
- *Capitalized words and their frequency.*
- *Hyphenated words and their frequency.*
- *Punctuation usage*: including usage of hyphens: - vs. – vs. —.
- *Choice of acceptable uses of words or phrases*: That is, choices taken when there is more than one alternative such as, British vs. American spelling and "I'm" vs. "I am".

The second category termed as *semantic information*, captures the essence of concepts being conveyed and the manner in which language is being used to convey these concepts.

- *Function word usage and their frequency of occurrence*: Function words are specific English words that are commonly used to convey ideas and their usage is believed to be independent of the ideas being conveyed. The frequency of usage of these words has been used with remarkable success in authorship attribution studies [14, 4]. For example, the function words used in our experiments are given in Appendix A.
- *Concepts, topics and specialized areas of interest*: This encompasses the areas of expertise, the knowledge, the presentation style and the reasoning process of the author. This is the realm of traditional literature studies and it is unclear how computers can help in this arena.

A *profile* of a user consists of the some or all of the above syntactic and semantic information. An example profile of a user of the *sci.crypt* newsgroup who posted 554 articles over a period of 5 years is given in Appendix B. Intuitively, each of these attributes appears to be dependent on an author's style of writing and communication. One can analyze each of these attributes and study their effectiveness in determining the identity behind a pseudonym. Such an analysis falls within the purview of the field of stylometry. However, text collected from the Internet setting contains several additional attributes (such as misspellings, lines between paragraphs etc) typically not present in the type of text studied by classical stylometry.

3 Analysis of the Usage of Function Words

The current state of art in stylometry has achieved considerable success by using the *principal component analysis* technique from statistics on function word usage [4]. In particular, it has succeeded in attributing authorship in several disputed works.

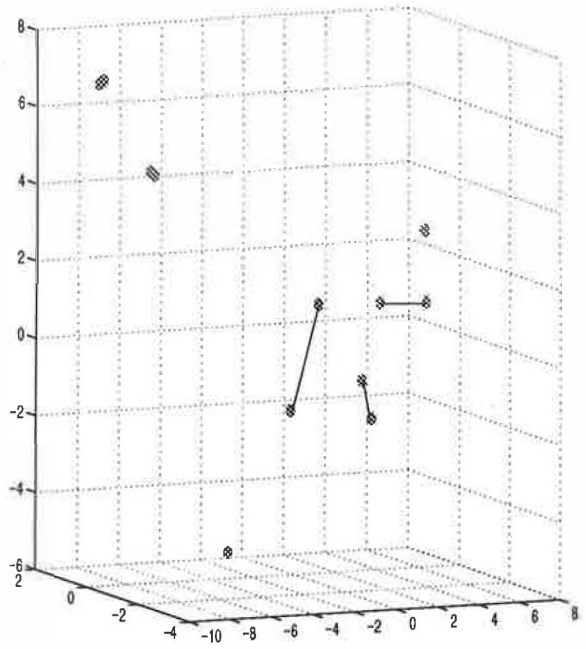
Our experiments show that this technique would be extremely effective in breaking pseudonymity in newsgroup postings. What was surprising to us was that even though the linguists have mostly used this technique to decide between two (or few) disputed authors, the technique is powerful enough to decompose a moderately sized (say of size 100) collection of pseudonyms into clusters where each cluster is highly likely to correspond to postings from a single identity.

The starting point for this technique is the computation of *frequency counts* of function words for each pseudonym. As mentioned earlier, function words are specific English words that are commonly used to convey ideas and are independent of the ideas being conveyed. They were defined and used with great success in one of the most celebrated works of statistical authorship attribution: the attribution of the 12 disputed *Federalist* papers to Madison [14, 15].

The frequency counts are then divided by the total number of words, to get an estimate of the probability of usage of each of these function words by the pseudonym. In our experiment, we also computed the estimate of the probability that the pseudonym uses a word outside the list. Traditionally, these probability estimates were used directly in authorship attribution studies [14], however direct analysis of function word usage is complicated by the fact that, from a statistical view, usage of different function words is not independent and this dependence should be accounted for in the analysis.

One way to view these probability estimates is as a vector space over the reals, where the dimensionality of the vector space corresponds to the number of function words being analyzed. Each author can be identified by a *author–point* in this vector space, i.e., the i'th co-ordinate of the author-point is just the the probability estimate of the author using the i'th function word. For authorship attribution studies, a direct use of the author-points defined with respect

Figure 1: Plot of 12 author-points along the first 3 principal components



this co-ordinate system has problems since the observed distributions of author-points along different axes are not going to be independent, since there is dependence between the usage of different function words.

A major breakthrough in finding a better co-ordinate system for this problem was developed by Burrows [4], who applied the principal component analysis approach from statistics to the collection author–points themselves. The principal component analysis approach when applied to the given author–points yields a new set of orthogonal basis vectors (or axes) for the vector space, which have the property the distributions of author–points along the new axes are *statistically independent.* These axes are known as principal components and furthermore these principal components can ordered on the basis of decreasing discriminating capabilities, i.e., the variation within the author-points is maximum along the first axis (or component) and minimum along the last axis (or component). Thus, by transforming the author–points to a representation with respect to these new axes or principal components, we get a collection of transformed points for which we can use very simple measures to gauge similarity. Moreover, since most of the discriminating information or variation in the transformed author-point data set is in the first few components, similarity analysis can be further restricted to them. A very simple similarity measure could be the euclidean distance between the transformed author-points. Alternatively, we can treat these transformed points as vectors and define a similarity measure between two vectors as the cosine of the angle between them. This measure is fairly easy to compute since it is just the inner-product of the normalized form of the two vectors and this measure has been extensively used in text processing and information retrieval systems. Yet another similarity measure could be the correlation between the various co-ordinates of the transformed author points. In our experiments and analysis we chose the simplest similarity measure, i.e., euclidean distance, which yielded fairly good results. We also experimented with the other measures and the results were equally good.

The effectiveness of this approach is illustrated in Figure 1 which plots the representation of 12 author-points (the first 12 author points out of a collection of 126 authors in one of our experiments) along the top 3 principal components. These 12 points actually consisted of only 7 distinct authors with 5 of these authors writing under 2 distinct pseudonyms. Even with 3 components (or dimensions), it is easy to identify two of these multiple pseudonym authors since the corresponding author-points are in very close proximity. We have also marked the other 3 multiple pseudonym authors by connecting the author points of their pseudonyms by a line. At first glance, it appears that these pairs of author points are not sufficiently close to be immediately identified as pseudonyms. However, it should be noted that typically, the number of independent dimensions we consider is around 50 and such a high dimension space, the author-points corresponding same identity are likely to be close in most dimensions and author-points corresponding to two different identities are likely to be far apart in several dimensions and this difference in behavior can be easily distinguished by the similarity measures described above.

Technically, the principal component analysis approach begins by defining a $W \times A$ matrix M, where the rows correspond to each one of the W function words and the columns correspond to each of the A authors. A requirement is that $W < A$. Each (w, a) entry in M corresponds to the estimate of probability of usage of word w by author a. Note that an author–point, as defined above, corresponds to

a column of this matrix. The first step in the process is to normalize the matrix M by ensuring that each row has zero mean and unit variance. This can be done by subtracting the row average from each entry and dividing the result by the row's standard deviation. The reason for this normalization is to ensure that each dimension (or function word) is treated equally in the subsequent processing, since, a priori, we have no reason assign different weights to different function words. However, in more specialized scenarios, one may be able to obtain better results if all the dimensions are not equally weighed.

Next, compute the $W \times W$ co–variance matrix C as follows:

$$C = \frac{1}{A} M \times M^t$$

where M^t is the transpose matrix. Intuitively, the co–variance matrix is the "column–to–column" similarity matrix having high values c_{ij} if columns i and j are highly correlated. Note that since we had normalized the matrix M, C is also the correlation co-efficient matrix for the usage of the W function words by the A authors. In specialized scenarios M may not be normalized and the co-variance matrix C will not be the correlation matrix.

The next step is to compute the eigenvalues and the corresponding eigenvectors of the co–variance matrix C. One can easily verify that the distinct eigenvectors are orthogonal over the real space. In fact, these eigenvectors form the principal components for the given data set. Furthermore, the larger the eigenvalue, the larger the utility of the corresponding eigenvector in discriminating between the different author–points. This is because, the variance of author–points when projected along an eigenvector is proportional to the corresponding eigenvalue. The final step is to obtain the new representation of each of the author–points with respect to the principal components or eigenvectors. Note that it is enough to restrict the representation for each author–point to those eigenvectors that have significant variance and hence significant discriminatory information. This can be accomplished by pre–multiplying M by a matrix whose rows consist of only the significant eigenvectors.

Principal component analysis is a standard tool in several mathematical toolkits such as Matlab [11].

4 Experiments

To explore and illustrate our ideas, we began by collecting three types of easily accessible, commonly archived and singly authored documents, namely, the *sci.crypt* newsgroup, the IPSec mailing list and the RFC Database.

Each type of document has a standard author identifier field (e.g., *From:* header in e–mails and news articles). We observed that, in practice, many authors used somewhat different identifiers (such as slightly different names, different e–mail addresses etc) over time. As a first step, we used elementary heuristics to aggregate each of the slightly different identifiers (and the respective documents) as corresponding to an entity that we call a *persona*. This was done so that we could obtain a much larger collection of documents per persona, in order to get sufficient data for statistical analysis [1].

After extracting personas, we removed all identifying information (such as headers, signatures etc) from the documents. At this point, our database of documents and associated personas is no different from what we would have obtained if we had completely pseudonymized documents. Once we partitioned the document space on a persona basis, it is easy to simulate multiple pseudonyms per persona by sub–partitioning the document space of each persona.

In the following sub–sections, we discuss two experiments that we performed on this pool of documents and the results obtained. Out of respect for the privacy rights of the concerned personas we have chosen not to name them in this paper.

4.1 Clustering Newsgroup Persona

Our initial data set consisted of 53914 articles available from the USENET archives of *sci.crypt* (articles # 6871 to # 62874 at deja-news.com) covering the period from 5 Jan 1992 to 28 Feb 1997. There were over 10758 personas who posted articles over this period. As mentioned above, each article was cleaned by removing headers and sig-

[1] As will become evident later, even if we had not done so, our tools would have identified multiple identifiers as belonging to the same persona. In fact, our tools led us to discover multiple gaps in our persona aggregation heuristics.

natures. Further heuristics were used to remove quoted/referenced text and non–language words.

Suppose we had a set of newsgroup postings by multiple persons each of whom used a small set of different pseudonyms for their postings. For a technique to break pseudonymity, it should be effective at clustering pseudonyms based solely on the content of the postings. Clearly, it is reasonable to apply such a technique to pseudonyms with a minimum but reasonable number of articles/words, since even the best technique would fail on very little data.

Testing the effectiveness of the principal component analysis technique to break pseudonymity in newsgroup postings can be easily simulated with the data we have. We only keep personas which meet a minimum requirement on articles/words. We create multiple pseudonyms per persona (for instance, two pseudonyms for a persona) by partitioning the document set of those personas who have a large number of articles/words. We can then apply the principal components technique to the collection of postings corresponding to these pseudonyms to obtain a principal component representation of author–points corresponding to these pseudonyms. Clustering of pseudonyms could based on the euclidean distances between the principal component representations of author–points. For example, in the two way split, two pseudonyms are assumed to correspond to the same person, if both are mutually closest to one another in the principal component representation. For more than two pseudonyms per person more advanced clustering metrics can be used.

- *Experiment I*: As a first experiment, we considered only those personas who had posted more than 50 articles and more that 5000 words. There were 117 such personas accounting for 19415 articles. Of these, we could chronologically split postings from 68 personas into two pseudonyms, the "earlier postings" pseudonym the "later postings" pseudonym, while still maintaining the article/word requirements[2]. This resulted in 185 pseudonyms, in which 68 personas had two pseudonyms and the remaining 49 had a single pseudonyms.

  Results: The principal component analysis technique was able to correctly cluster 40 out of 68 multiple pseudonym personas and reported 2 false positives, i.e., 2 pairs wrongly classified as a single identity. This gives a success rate of 58.8% and a false positive rate [3] of 2.2%.

- *Experiment II*: As a second experiment we considered personas with more than 50 articles and more that 10000 words. There were 84 such personas accounting for 17001 articles. Of these, we were able to chronologically split 42 personas into two pseudonyms. This resulted in 126 pseudonyms, in which 42 personas had two pseudonyms and the remaining 42 had a single pseudonyms.

  Results: The principal component analysis technique was able to correctly cluster 34 out of 42 multiple pseudonym personas and reported 1 false positive. This gives a success rate of 80.9% and a false positive rate of 1.6%.

We also experimented with varying the minimum requirements on the number of words and found that at around 6500 words the results are almost as good as for 10000 words. It appears that this is a reasonable threshold above which there is significant leakage of identity information.

In many false negative cases, the closest neighbor relationship held for one of the points and not for the other, and in most false negative cases the pairs were amongst the top few closest points to each other. This suggests that a slightly relaxed clustering criterion in conjunction with other, perhaps, syntactic criteria may yield better results.

Also note that the pair–clustering criterion we have chosen, i.e., mutually closest points in a high dimensional space, will usually find significant number of pairs in most cases, e.g., even when points are randomly chosen. Therefore, if there are very few multiple pseudonyms in the data set, we would get a significant false positive rate. In such cases a different approach would be better. For example, one could consider only the suspected multiple pseudonyms and look for author points closest to them. Another approach would be to do further processing on the clustered pairs, by looking at the nature of the proximity of the pair (e.g., the value of the distance vs. average distance when one of the identities in the pair is arbitrarily split, distribution

[2] We had several options on how the set of articles of a persona could be split. For example, we could have split the articles randomly. However, we chose to split chronologically since it is well known that an author's style evolves over time, and such a split should be slightly harder to reconcile

[3] Computed as the number of pseudonyms wrongly paired divided by the total number of pseudonyms.

of author points in the neighborhood of the pair etc) or perhaps by employing other syntactic and semantic criteria to weed out the false positives.

4.2 Linking E–mail Persona to Newsgroup Persona

One can argue that our results on breaking pseudonymity in newsgroup are not applicable, since people may not use more than one pseudonym for a single activity such as posting to newsgroups. For example, the FAQ from ZeroKnowledge.com states that one can use a different "nym" for each different activity. To show the generality of our approach, we choose a second domain, namely postings to mailing list and show how a persona in this domain can be linked to the corresponding newsgroup persona.

The first set of experiments described above, enabled us to build statistical profiles of frequent sci.crypt posters. To test whether a mailing list persona could be linked to a newsgroup persona, we looked for examples of personas who have posted material to both domains. In particular, we examined the list of frequent contributors to the IPSec mailing list archives from Aug 14, 1992 to Dec 31 1996 and found that there were 7 contributors who had posted sufficient material in both forums. As before, we preprocessed the IPSec archive to extract the e–mails of these 7 personas.

For each of these personas, we performed a principal component analysis test (as described earlier) on the frequent (> 10000 words) sci.crypt personas and the e–mail personas. We then looked at the proximity of the given e–mail persona to the sci.crypt personas. In 5 of the 7 cases, the corresponding newsgroup persona was the closest point to the e–mail persona. In one case, the corresponding newsgroup persona was second closest and in the remaining case, the corresponding personas were far apart.

In a sense, these results are not particularly surprising and merely reinforce the analysis for the newsgroup postings. It is also intuitive to expect this since e–mail postings are very similar in style to newsgroup postings. One could expect similar results to hold for chat–room transcripts.

4.3 A Comment on Styles: RFCs vs. Newsgroup Postings

We tried to extend the results of the previous section to other domains by comparing RFC personas with newsgroup personas. Preliminary experiments indicate that the linking of the personas in this case is quite poor. We suspect that the reason for this is that the writing style for newsgroup posting/e–mail is quite informal and direct, whereas RFC writing style is much more formal and indirect. We speculate that it may be feasible to find a transformation which would map the style of RFC personas to Newsgroup personas.

5 Future Directions

During the course of this work, we found several intriguing leads and potential directions for future research. Most promising amongst these is the wealth of identifying information that is available in the syntactic aspects of internet based communications. While we have already enumerated the various useful syntactic features before, it is important to note that we did not use any of this information in the analysis presented in the previous section. A number of these features have also not been studied in classical authorship studies since the latter have dealt mostly with proof-read texts that have appeared in well-scrutinized and standard formats traditionally followed by publishers. For example identifying features like misspellings, types of formatting (spacing, paragraph indentation style etc) and other stylistic idiosyncrasies such as hyphenation style, capitalization style, etc, are characteristic of internet based communications but rare in published literature.

We now detail some of our initial findings in studying the discriminating potential of these syntactic features of internet based communications. A major discriminating feature appears to be the set of words misspelled by a persona. We observed that a majority of the documents we dealt with contained misspellings. Broadly speaking, there are three types of misspellings: The most important type for our purposes are words which are consistently misspelled by an author. We have observed that such a list of words can be used as a good discriminator for the author. In fact, very often this list could consist

of a few or even a single word. For example, three prominent sci.crypt posters could be easily identified by their use of the words "naiive", "cought & propietary" and "consistantly & insistance" respectively.

Closely related to this category of misspelling is the usage of British vs. American English spelling of words. Use of words such "authorise", "behaviour", "centre", "colour", for example, would clearly place the author in the British English camp.

The second type of misspelling are those words of which the author is somewhat unsure and are misspelled with some probability. These probabilities could potentially be used for discrimination purposes. The third type of misspelling are typos. These may be indicative of the author's typing style. More work needs to be done to establish the significance of these two categories.

Our preliminary analysis also indicates that formatting features such as spacing between paragraphs, paragraph indentation style, capitalized words, hyphenated words and words with an apostrophe, also have good discrimination potential.

In passing, we would like to mention the role of other syntactic features that have been studied in by classical stylometry such as vocabulary, choice of specific words to convey concepts and punctuation style in authorship attribution.

We speculate that a hybrid approach which uses the principal component analysis approach on function words and formatting features, coupled with misspelling lists and the classical stylometry measures mentioned above would be extremely effective in attacking pseudonymity.

6 Possible Countermeasures

In this paper, we have described two classes of information leakage: syntactic and semantic, both of which can be used for attacking pseudonymity.

Countermeasures should try to address both classes of leakage. For each class and for each type of leakage within the class, a countermeasure should either eliminate the leakage altogether or substantially reduce the amount of information leaked so that an adversary is unlikely to have access to the minimum amount of text needed to apply the attacks. On the other hand, drastic countermeasures run the risk of badly altering the meaning of the document thereby requiring extensive manual intervention and rendering them unusable.

It should be easy to design minimally disruptive countermeasures to tackle most syntactic leakage. However, fixing the problem of vocabulary requires more investigation. For instance a thesaurus tool could prompt the user to consider alternatives while composing messages, thereby reducing variations in vocabulary.

However, countermeasures to address semantic leakage appear to be hard to design. One drastic (and somewhat facetious) approach is to use natural language translation systems to translate a document from English to another language and back. While this may destroy the function word frequencies in the original document, it would considerably change the meaning of the message, given the primitive state of translation systems. Even this would not destroy information about the meta–level concepts that an author uses in his documents.

Apart from these automated countermeasures, another powerful countermeasure (as suggested to us by an anonymous referee) would be to educate users of pseudonymity services about the types of exposures and attacks discussed in this paper. A user community which is aware of these types of exposures may be the most successful way to minimize the effectiveness of these attacks.

7 Acknowledgements

The authors would like to thank Roy Byrd, Suresh Chari, Pau-Chen Cheng, Charanjit Jutla, Aaron Kerschenbaum and Charles Palmer for several useful insights and helpful discussions. We would also like to thank anonymous referees for their comments and suggestions on improving this paper.

References

[1] The Anonymizer Service, `www.anonymizer.com`.

[2] Robert A. Bosch and Jason A. Smith, "Separating Hyperplanes and the Authorship of Disputed Federalist Papers", The American Mathematical Monthly, Volume 105, Number 7, August–September 1998, pages 601–608.

[3] Barron Brainerd, "The Computer in Statistical Studies of William Shakespeare", Computer Studies in the Humanities and Verbal Behavior, Volume 4, Number 1, pages 9–15, 1973.

[4] J.F. Burrows, "Word Patterns and Story Shapes: The Statistical Analysis of Narrative Style", Literary and Linguistic Computing, Volume 2, pages 61–70, 1987.

[5] Lorrie Faith Cranor, "Internet Privacy", Communications of the ACM, Volume 42, Number 2, February 1999, pages 29–31.

[6] Alvar Ellegard, "A Statistical Method for Determining Authorship: the Junius Letters 1769–1772", Gothenburg Studies in English No. 13, Acta Universitatis Gothenburgensis, Elanders Boktryckeri Aktiebolag, Goteborg.

[7] Evan Gabber, Phillip B. Gibbons, David M. Kristol, Yossi Matias and Alain Mayer, "Consistent, Yet Anonymous, Web Access with LPWA", Communications of the ACM, Volume 42, Number 2, February 1999, pages 42–47.

[8] David Goldschlag, Michael Reed and Payl Syverson, "Onion Routing for Anonymous and Private Internet Connections", Communications of the ACM, Volume 42, Number 2, February 1999, pages 39–41.

[9] `www.freedom.net`.

[10] `www.freedom.net/info/freedompapers/Freedom-Architecture.pdf`.

[11] `www.mathworks.com`.

[12] T.C. Mendenhall, "A Mechanical Solution to a Literary Problem", Popular Science Monthly, Volume 60, Number 2, pages 97–105, 1901.

[13] A.Q. Morton, "The Authorship of Greek Prose", Journal of the Royal Statistical Society (A), Volume 128, pages 169–233, 1965.

[14] Frederick Mosteller and David L. Wallace, "Inference and Disputed Authorship: The Federalist", Addison–Wesley, Reading, MA, 1964.

[15] Frederick Mosteller and David L. Wallace, "Applied Bayesian and Classical Inference: The Case of The Federalist Papers", Springer Series in Statistics, Springer–Verlag, 1984.

[16] Joseph Reagle and Lorrie Faith Cranor, "The Platform for Privacy Preferences", Communications of the ACM, Volume 42, Number 2, February 1999, pages 48–55.

[17] Michael K. Reiter and Aviel D. Rubin, "Anonymous Web Transactions With Crowds", Communications of the ACM, Volume 42, Number 2, February 1999, pages 32–38.

[18] C.B. Williams, "Mendenhall's Studies of Word-Length Distribution in the Works of Shakespeare and Bacon", Biometrika, Volume 62, pages 207–212, 1975.

A Function Words

a	*able*	*about*	*above*
after	*again*	*against*	*all*
allow	*almost*	*also*	*although*
am	*among*	*an*	*and*
another	*any*	*apply*	*are*
as	*ask*	*at*	*away*
be	*because*	*been*	*being*
between	*but*	*by*	*can*
could	*did*	*do*	*does*
down	*either*	*enough*	*even*
every	*for*	*from*	*get*
give	*had*	*has*	*have*
he	*her*	*his*	*how*
i	*if*	*in*	*into*
is	*it*	*make*	*may*
me	*might*	*more*	*most*
must	*my*	*no*	*nor*
not	*now*	*of*	*often*
on	*one*	*only*	*or*
other	*our*	*same*	*say*
see	*shall*	*she*	*should*
so	*some*	*someone*	*still*
such	*sure*	*take*	*than*
that	*the*	*their*	*them*
then	*there*	*they*	*this*
to	*under*	*until*	*up*
upon	*use*	*want*	*was*
we	*were*	*what*	*when*
where	*which*	*who*	*why*
will	*with*	*would*	*yes*
you	*your*		

B Profile of a Persona

```
Persona: 9069
Number of Articles: 554
Total Number of Words: 91257

1. Blank Line Distribution:

1: 2418
2: 768
.
.
.
123: 1
359: 1

Mean: 1.913 Variance: 39.980

2. Word Count Distribution:

0: 3
2: 1
.
.
.
4372: 1

Mean: 164.724 Variance: 180411.828

3. Number of Paragraphs Distribution:

1: 4
2: 102
.
.
.
252: 1

Mean:7.863 Variance: 406.642

4. Paragraph Size Distribution:

1: 5
2: 253
.
.
.
606: 1
776: 1

Mean: 20.855 Variance: 749.314

5. Paragraph Indentation Style:

Does not use tabs to indent
paragraphs:

Used spaces to indent paragraphs:

Mean: 5.023 Variance: 29.718

6. Hyphenation styles:

 - (283)  -- (21)  --- (2)

7. 9069 uses the following words
   commonly:

a: 1981
```

```
about: 174
.
.
.
with: 500
you: 292

8. 9069 uses apostrophes in the
   following words:

Alice's, Bamford's, Bidzos', ..., He'd,
..., hoaxer's, ..., you're, you've.

9. 9069 likes to completely capitalize
   the following words:

ACCESS, ACLU, ACM, ..., YOU, YOUR, ZIP

10. 9069's vocabulary (including
    misspelled words):

ACCESS, ACLU, ACM, ..., Zero-Knowledge,
...,  ability, able, ..., zeroize,
zeros.

11. 9069's misspelled words:

ableto, accessdata, ...,
cought, ..., yeild
```

An Open-source Cryptographic Coprocessor

Peter Gutmann
University of Auckland, Auckland, New Zealand
pgut001@cs.auckland.ac.nz

Abstract

Current crypto implementations rely on software running under general-purpose operating systems alongside a horde of untrusted applications, ActiveX controls, web browser plugins, mailers handling messages with embedded active content, and numerous other threats to security, with only the OS's (often almost nonexistant) security to keep the two apart. This paper presents a general-purpose open-source crypto coprocessor capable of securely performing crypto operations such as key management, certificate creation and handling, and email encryption, decryption, and signing, at a cost one to two orders of magnitude below that of commercial equivalents while providing generally equivalent performance and a higher level of functionality. The paper examines various issues involved in designing the coprocessor, and explores options for hardware acceleration of crypto operations for extended performance above and beyond that offered by the basic coprocessor's COTS hardware.

1. Problems with Crypto on End-user Systems

The majority of current crypto implementations run under general-purpose operating systems with a relatively low level of security, alongside which exist a limited number of smart-card assisted implementations which store a private key in, and perform private-key operations with, a smart card. Complementing these are an even smaller number of implementations which perform further operations in dedicated (and generally very expensive) hardware.

The advantage of software-only implementations is that they are inexpensive and easy to deploy. The disadvantage of these implementations is that they provide a very low level of protection for cryptovariables, and that this low level of security is unlikely to change in the future. For example Windows NT provides a function `ReadProcessMemory` which allows a process to read the memory of (almost) any other process in the system (this was originally intended to allow debuggers to establish breakpoints and maintain instance data for other processes [1]), allowing both passive attacks such as scanning memory for high-entropy areas which constitute keys [2] and active attacks in which a target processes' code or data is modified (in combination with `VirtualProtectEx`, which changes the protection on another processes' memory pages) to provide supplemental functionality of benefit to a hostile process. By subclassing an application such as the Windows shell, the hostile process can receive notification of any application (a.k.a. "target") starting up or shutting down, after which it can apply the mechanisms mentioned previously. A very convenient way to do this is to subclass a child window of the system tray window, yielding a system-wide hook for intercepting shell messages [3]. Another way to obtain access to other processes' data is to patch the user-to-kernel-mode jump table in a processes' Thread Environment Block (TEB), which is shared by all processes in the system rather than being local to each one, so that changing it in one process affects every other running process [4].

Although the use of functions like `ReadProcessMemory` requires Administrator privileges, most users tend to either run their system as Administrator or give themselves equivalent privileges since it's extremely difficult to make use of the machine without these privileges. In the unusual case where the user isn't running with these privileges, it's possible to use a variety of tricks to bypass any OS security measures which might be present in order to perform the desired operations. For example by installing a Windows message hook it's possible to capture messages intended for another process and have them dispatched to your own message handler. Windows then loads the hook handler into the address space of the process which owns the thread which the message was intended for, in effect yanking your code across into the address space of the victim [5]. Even simpler are mechanisms such as using the HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs key, which specifies a list of DLLs which are automatically loaded and called whenever an application uses the USER32 system library (which is automatically used by all GUI applications and many command-line ones). Every DLL specified in this registry key is loaded into the processes' address space by USER32, which then calls the DLL's `DllMain` function to initialise the DLL (and, by extension, trigger whatever other actions the DLL is designed for).

A more sophisticated attack involves persuading the system to run your code in ring 0 (the most privileged security level usually reserved for the OS kernel) or, alternatively, convincing the OS to allow you to load a selector which provides access to all physical memory (under Windows NT, selectors 8 and 10 provide this capability). Running user code in ring 0 is possible due to the peculiar way in which the NT kernel loads. The kernel is accessed via the int 2Eh call gate, which initially provides about 200 functions via NTOSKRNL.EXE but is then extended to provide more and more functions as successive parts of the OS are loaded. Instead of merely adding new functions to the existing table, each new portion of the OS which is loaded takes a copy of the existing table, adds its own functions to it, and then replaces the old one with the new one. To add supplemental functionality at the kernel level, all that's necessary is to do the same thing [6]. Once your code is running at ring 0, an NT system starts looking a lot like a machine running DOS.

Although the problems mentioned so far have concentrated on Windows NT, many Unix systems aren't much better. For example the use of `ptrace` with the PTRACE_ATTACH option followed by the use of other `ptrace` capabilities provides similar headaches to those arising from `ReadProcessMemory`. The reason why these issues are more problematic under NT is that users are practically forced to run with system Administrator privileges in order to perform any useful work on the system, since a standard NT system has no equivalent to Unix's `su` functionality and, to complicate things further, frequently assumes that the user always has Administrator privileges (that is, it assumes it's a single-user system with the user being Administrator). While it's possible to provide some measure of protection on a Unix system by running crypto code as a daemon in its own memory space, the fact that the Administrator can dynamically load NT services (which can use `ReadProcessMemory` to interfere with any other running service) means that even implementing the crypto code as an NT service provides no escape.

1.1. The Root of the Problem

The reason why problems like those described above persist, and why we're unlikely to ever see a really secure consumer OS is because it's not something which most consumers care about. One recent survey of Fortune 1000 security managers showed that although 92% of them were concerned about the security of Java and ActiveX, nearly three quarters allowed them onto their internal networks, and more than half didn't even bother scanning for them [7]. Users are used to programs malfunctioning and computers crashing (every Windows NT user can tell you what the abbreviation BSOD means even though it's never actually mentioned in the documentation), and see it as normal for software to contain bugs. Since program correctness is difficult and expensive to achieve, and as long as flashiness and features are the major selling point for products, buggy and insecure systems will be the normal state of affairs [8]. Unlike other Major Problems like Y2K (which contain their own built-in deadline), security generally isn't regarded as a pressing issue unless the user has just been successfully attacked or the corporate auditors are about to pay a visit, which means that it's much easier to defer addressing it to some other time [9]. Even in cases where the system designers originally intended to implement a rigorous security system employing a trusted computing base (TCB), the requirement to add features to the system inevitably results in all manner of additions being crammed into the TCB, with the result that it is neither small, nor verified, nor secure.

An NSA study [10] lists a number of features which are regarded as "crucial to information security" but which are absent from all mainstream operating systems. Features such as mandatory access controls which are mentioned in the study correspond to Orange Book B-level security features which can't be bolted onto an existing design but generally need to be designed in from the start, necessitating a complete overhaul of an existing system in order to provide the required functionality. This is often prohibitively resource-intensive, for example the task of reengineering the Multics kernel (which contained a "mere" 54,000 lines of code) to provide a minimised TCB was estimated to cost $40M (in 1977 dollars) and was never completed [11]. The work involved in performing the same kernel upgrade or redesign from scratch with an operating system containing millions or tens of millions of lines of code would make it beyond prohibitive.

At the moment security and ease of use are at opposite ends of the scale, and most users will opt for ease of use over security. JavaScript, ActiveX, and embedded active content may be a security nightmare, but they do make life a lot easier for most users, leading to comments from security analysts like "You want to write up a report with the latest version of Microsoft Word on your insecure computer or on some piece of junk with a secure computer?"[12], "Which sells more products: really secure software or really easy-to-use software?"[13], and "It's possible to make money from a lousy product [...] Corporate cultures are focused on money, not product"[14]. In many cases users don't even have a choice, if they can't process data from Word, Excel, PowerPoint, and Outlook and view web pages loaded with JavaScript and ActiveX, their business doesn't run, and some companies go so far as to publish explicit instructions telling users how to

disable security measures in order to maximise their web-browsing experience [15]. Going beyond basic OS security, most current security products still don't effectively address the problems posed by hostile code such as trojan horses (which the Orange Book's Bell-LaPadula security model was designed to combat), and the systems the code runs on increase both the power of the code to do harm and the ease of distributing the code to other systems.

This presents rather a gloomy outlook for someone wanting to provide secure crypto services to a user of these systems. In order to solve this problem, we adopt a reversed form of the Mohammed-and-the-mountain approach: Instead of trying to move the insecurity away from the crypto through various operating system security measures, we instead move the crypto away from the insecurity. In other words although the user may be running a system crawling with rogue ActiveX controls, macro viruses, trojan horses, and other security nightmares, none of these can come near the crypto.

1.2. Solving the Problem

The FIPS 140 standard provides us with a number of guidelines for the development of cryptographic security modules. NIST originally allowed only hardware implementations of cryptographic algorithms (for example the original NIST DES document allowed for hardware implementation only [16][17]), however this requirement was relaxed somewhat in the mid-1990's to allow software implementations as well [18][19]. FIPS 140 defines four security levels ranging from level 1 (the cryptographic algorithms are implemented correctly) through to level 4 (the module or device has a high degree of tamper-resistance including an active tamper response mechanism which causes it to zeroise itself when tampering is detected). To date only one general-purpose product family has been certified at level 4 [20].

Since FIPS 140 also allows for software implementations, an attempt has been made to provide an equivalent measure of security for the software platform on which the cryptographic module is to run. This is done by requiring the underlying operating system to be evaluated at progressively higher Orange Book levels for each FIPS 140 level, so that security level 2 would require the software module to be implemented on a C2-rated operating system. Unfortunately this provides something if an impedance mismatch between the actual security of hardware and software implementations, since it implies that products such as a Fortezza card [21] or Dallas iButton (a relatively high-security device) [22] provide the same level of security as a program running under Windows NT. It's possible that the OS security levels were set so low out of concern that setting them any higher would make it impossible to implement the higher FIPS 140 levels in software due to a lack of systems evaluated at that level.

Even with sights set this low, it doesn't appear to be possible to implement secure software-only crypto on a general-purpose PC. Trying to protect cryptovariables (or more generically security-relevant data items, SRDI's in FIPS 140-speak) on a system which provides functions like `ReadProcessMemory` seems pointless, even if the system does claim a C2/E2 evaluation. On the other hand trying to source a B2 or more realistically B3 system to provide an adequate level of security for the crypto software is almost impossible (the practicality of employing an OS in this class, whose members include Trusted Xenix, XTS 300, and Multos, speaks for itself). A simpler solution would be to implement a crypto coprocessor using a dedicated machine running at system high, and indeed FIPS 140 explicitly recognises this by stating that the OS security requirements only apply in cases where the system is running programs other than the crypto module (to compensate for this, FIPS 140 imposes its own software evaluation requirements which in some cases are even more arduous than the Orange Book ones).

An alternative to a pure-hardware approach might be to try to provide some form of software-only protection which attempts to compensate for the lack of protection present in the OS. Some work has been done in this area involving the obfuscation of the code to be protected, either mechanically [23] or manually [24]. The use of mechanical obfuscation (for example reodering of code and insertion of dummy instructions) is also present in a number of polymorphic viruses, and can be quite effectively countered [25][26]. Manual obfuscation techniques are somewhat more difficult to counter automatically, however computer game vendors have trained several generations of crackers in the art of bypassing the most sophisticated software protection and security features they could come up with [27][28][29], indicating that this type of protection won't provide any relief either, and this doesn't even go into the portability and maintenance nightmare which this type of code presents (it is for these reasons that the obfuscation provisions were removed from a later version of the CDSA specification where they were first proposed [30]).

1.3. Coprocessor Design Issues

The main consideration when designing a coprocessor to manage crypto operations is how much functionality we should move from the host into the coprocessor unit.

The baseline, which we'll call a tier[1] 0 coprocessor, has all the functionality in the host, which is what we're trying to avoid. The levels above tier 0 provide varying levels of protection for cryptovariables and coprocessor operations, as shown in Figure 1.

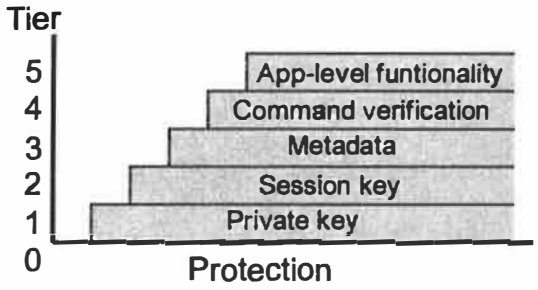


Figure 1: Levels of protection offered by crypto hardware

The minimal level of coprocessor functionality, a tier 1 coprocessor, moves the private key and private-key operations out of the host. This type of functionality is found in smart cards, and is only a small step above having no protection at all, since although the key itself is held in the card, all operations performed by the card are controlled by the host, leaving the card at the mercy of any malicious software on the host system. In addition to these shortcomings, smart cards are very slow, offer no protection for cryptovariables other than the private key, and often can't even protect the private key fully (for example a card with an RSA private key intended for signing can be misused to decrypt a key or message since RSA signing and decryption are equivalent).

The next level of functionality, tier 2, moves both public/private-key operations and conventional encryption operations along with hybrid mechanisms such as public-key wrapping of content-encryption keys into the coprocessor. This type of functionality is found in devices such as Fortezza cards and a number of devices sold as crypto accelerators, and provides rather more protection than that found in smart cards since no cryptovariables are ever exposed on the host. Like smart cards however, all control over the devices operation resides in the host, so that even if a malicious application can't get at the keys directly, it can still apply them in a manner other than the intended one.

The next level of functionality, tier 3, moves all crypto-related processing (for example certificate generation and message signing and encryption) into the coprocessor. The only control the host has over processing is at the level of "sign this message" or "encrypt this message", all other operations (message formatting, the addition of additional information such as the signing time and signers identity, and so on) is performed by the coprocessor. In contrast if the coprocessor has tier 1 functionality the host software can format the message any way it wants, set the date to an arbitrary time (in fact it can never really know the true time since it's coming from the system clock which another process could have altered), and generally do whatever it wants with other message parameters. Even with a tier 2 coprocessor such as a Fortezza card which has a built-in real-time clock (RTC), the host is free to ignore the RTC and give a signed message any timestamp it wants. Similarly, even though protocols like CSP which is used with Fortezza incorporate complex mechanisms to handle authorisation and access control issues [31], the enforcement of these mechanisms is left to the untrusted host system rather than the card(!). Other potential problem areas involve handling of intermediate results and composite call sequences which shouldn't be interrupted, for example loading a key and then using it in a cryptographic operation [32]. In contrast, with a tier 3 coprocessor which performs all crypto-related processing independent of the host the coprocessor controls the message formatting and the addition of additional inforation such as a timestamp taken from its own internal clock, moving them out of reach of any software running on the host. The various levels of protection when the coprocessor is used for message decryption are shown in Figure 2.

Going beyond tier 3, a tier 4 coprocessor provides facilities such as command verification which prevent the coprocessor from acting on commands sent from the host system without the approval of the user. The features of this level of functionality are explained in more detail in the section on extended security functionality.

Can we move the functionality to an even higher level, tier 5, giving the coprocessor even more control over message handling? Although it's possible to do this, it isn't a good idea since at this level the coprocessor will potentially need to run message viewers (to display messages), editors (to create/modify messages), mail software (to send and receive them), and a whole host of other applications, and of course these programs will need to be able to handle MIME attachments, HTML, JavaScript, ActiveX, and so on in order to function as required. In addition the coprocessor will now require its own input mechanism (a keyboard), output mechanism (a monitor), mass storage, and other extras. At this point the coprocessor has evolved into a second computer attached to the original one, and since it's running a range of untrusted and potentially dangerous

[1] The reason for the use of this somewhat unusual term is because almost every other noun used to denote hierarchies is already in use; "teir" is unusual enough that noone else has got around to using it in their security terminology.

code we need to think about moving the crypto functionality into a coprocessor for safety. Lather, rinse, repeat.

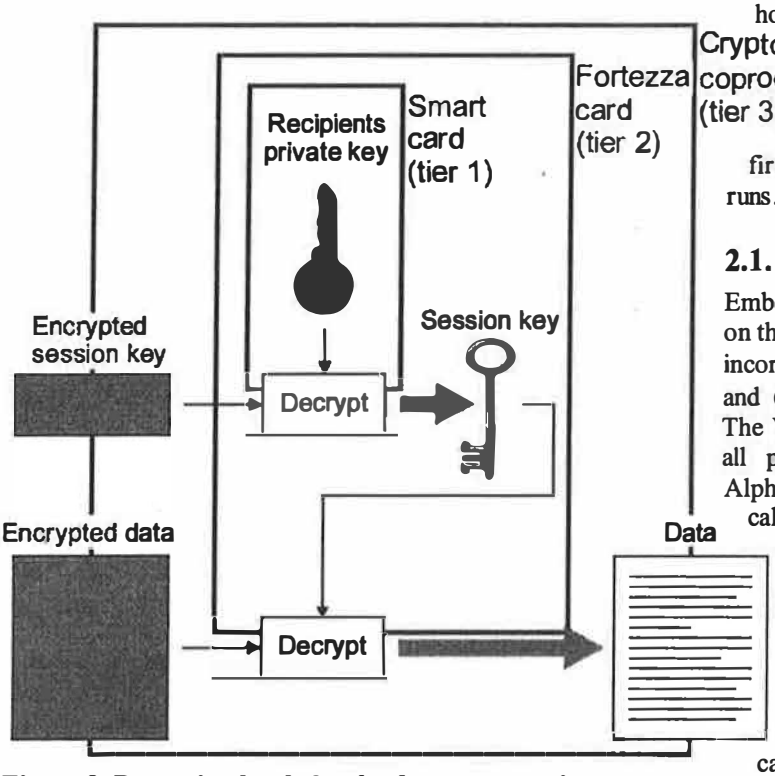


Figure 2: Protection levels for the decrypt operation

The best level of functionality therefore is to move all crypto and security-related processing into the coprocessor, but to leave everything else on the host.

2. The Coprocessor

The traditional way to build a crypto coprocessor has been to create a complete custom implementation, originally with ASIC's and more recently with a mixture of ASIC's and general-purpose CPU's, all controlled by custom software. This approach leads to long design cycles, difficulties in making changes at a later point, high costs (with an accompanying strong incentive to keep all design details proprietary due to the investment involved), and reliance on a single vendor for the product. In contrast an open-source coprocessor by definition doesn't need to be proprietary, so it can use existing COTS hardware and software as part of its design, which greatly reduces the cost (the coprocessor described here is one to two orders of magnitude cheaper than proprietary designs while offering generally equivalent performance and superior functionality), and can be sourced from multiple vendors and easily migrated to newer hardware as the current hardware base becomes obsolete.

The coprocessor requires three layers, the processor hardware, the firmware which manages the hardware (for example initialisation, communications with the host, persistent storage, and so on) and the Crypto software which handles the crypto coprocessor functionality. The following (tier 3) sections describe the coprocessor hardware and resource management firmware on which the crypto control software runs.

2.1. Coprocessor Hardware

Embedded systems have traditionally been based on the VME bus, a 32-bit data/32-bit address bus incorporated onto cards in the 3U (10×16cm) and 6U (23×16cm) Eurocard form factor [33]. The VME bus is CPU-independent and supports all popular microprocessors including Sparc, Alpha, 68K, and x86. An x86-specific bus called PC/104, based on the 104-pin ISA bus, has become popular in recent years due to the ready availability of low-cost components from the PC industry. PC/104 cards are much more compact at 9×9.5cm than VME cards, and unlike a VME passive backplane-based system can provide a complete system on a single card [34]. PC/104-Plus, an extension to PC/104, adds a 120-pin PCI connector alongside the existing ISA one, but is otherwise mostly identical to PC/104 [35]

In addition to PC/104 there are a number of functionally identical systems with slightly different form factors, of which the most common is the biscuit PC, a card the same size as a 3½" or occasionally 5¼" drive, with a somewhat less common one being the credit card or SIMM PC roughly the size of a credit card. A biscuit PC provides most of the functionality and I/O connectors of a standard PC motherboard, as the form factor shrinks the I/O connectors do as well so that a SIMM PC typically uses a single enormous edge connector for all its I/O. In addition to these form factors there also exist card PC's (sometimes called slot PC's), which are biscuit PC's built as ISA or (more rarely) PCI-like cards. A typical configuration for a low-end system is a 5x86/133 CPU (roughly equivalent in performance to a 133 MHz Pentium), 8-16MB of DRAM, 2-8MB of flash memory emulating a disk drive, and every imaginable kind of I/O (serial ports, parallel ports, floppy disk, IDE hard drive, IR and USB ports, keyboard and mouse, and others). High-end embedded systems built from components designed for laptop use provide about the same level of performance as a current laptop PC, although their price makes them rather impractical for use as crypto hardware. To

compare this with other well-known types of crypto hardware, a typical smart card has a 5MHz 8-bit CPU, a few hundred bytes of RAM, and a few kB of EEPROM, and a Fortezza card has a 10 or 20MHz ARM CPU, 64kB of RAM and 128kB of flash memory/EEPROM.

All of the embedded systems described above represent COTS components available from a large range of vendors in many different countries, with a corresponding range of performance and price figures. Alongside the x86-based systems there also exist systems based on other CPU's, typically ARM, Dragonball (embedded Motorola 68K), and to a lesser extent PowerPC, however these are available from a limited number of vendors and can be quite expensive. Besides the obvious factor of system performance affecting the overall price, the smaller form factors and use of exotic hardware such as non-generic-PC components can also drive up the price. In general the best price/performance balance is obtained with a very generic PC/104 or biscuit PC system.

2.2. Coprocessor Firmware

Once the hardware has been selected the next step is to determine what software to run on it to control it. The coprocessor is in this case acting as a special-purpose computer system running only the crypto control software, so that what would normally be thought of as the operating system is acting as the system firmware, and the real operating system for the device is the crypto control software. The control software therefore represents an application-specific operating system, with crypto objects such as encryption contexts, certificates, and envelopes replacing the user applications which are managed by conventional OS's. The differences between a conventional system and the crypto coprocessor running one typical type of firmware-equivalent OS are shown in Figure 3.

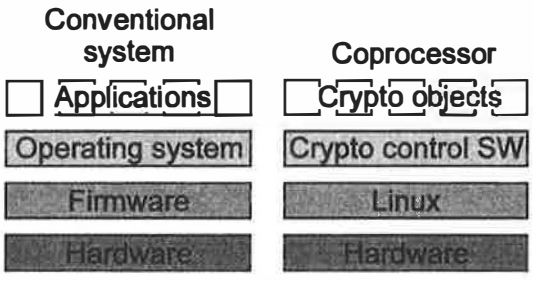


Figure 3: Conventional system vs. coprocessor system layers

Since the hardware is in effect a general-purpose PC, there's no need to use a specialised, expensive embedded or real-time kernel or OS since a general-purpose OS will function just as well. The OS choice is then something simple like one of the free or nearly-free embeddable forms of MSDOS [36][37][38] or an open source operating system like one of the x86 BSD's or Linux which can be adapted for use in embedded hardware. Although embedded DOS is the simplest to get going and has the smallest resource requirements, it's really only a bootstrap loader for real-mode applications and provides very little access to most of the resources provided by the hardware. For this reason it's not worth considering except on extremely low-end, resource-starved hardware (it's still possible to find PC/104 cards with 386/40's on them, although having to drive them with DOS is probably its own punishment).

A better choice than DOS is a proper operating system which can fully utilise the capabilities of the hardware. The only functionality which is absolutely required of the OS is a memory manager and some form of communication with the outside world. Also useful (although not absolutely essential) is the ability to store data such as private keys in some form of persistent storage. Finally, the ability to handle multiple threads may be useful where the device is expected to perform multiple crypto tasks at once. Apart from the multithreading, the OS is just acting as a basic resource manager, which is why DOS could be pressed into use if necessary.

Both FreeBSD and Linux have been stripped down in various ways for use with embedded hardware [39][40]. There's not really a lot to say about the two, both meet the requirements given above, both are open source systems, and both can use a standard full-scale system as the development environment — whichever one is the most convenient can be used. At the moment Linux is a better choice because its popularity means there's better support for devices such as flash memory mass storage (relatively speaking, as the Linux drivers for the most widely-used flash disk are for an old kernel while the FreeBSD ones are mostly undocumented and rather minimal), so the coprocessor described here uses Linux as its resource management firmware. A convenient feature which gives the free Unixen an extra advantage over alternatives like embedded DOS is that they'll automatically switch to using the serial port for their consoles if no video drivers and/or hardware are present, which enables them to be used with cheaper embedded hardware which doesn't require additional video circuitry just for the one-off setup process. A particular advantage of Linux is that it'll halt the CPU when nothing is going on (which is most of the time), greatly reducing coprocessor power consumption and heat problems.

2.3. Firmware Setup

Setting up the coprocessor firmware involves creating a stripped-down Linux setup capable of running on the coprocessor hardware. The services required of the firmware are:

- Memory management
- Persistent storage services
- Communication with the host
- Process and thread management (optional)

All newer embedded systems support the M-Systems DiskOnChip (DOC) flash disk, which emulates a standard IDE hard drive by identifying itself as a BIOS extension during the system initialisation phase (allowing it to install a DOC filesystem driver to provide BIOS support for the drive) and later switching to a native driver for OS's which don't use the BIOS for hardware access [41]. The first step in installing the firmware involves formatting the DOC as a standard hard drive and partitioning it prior to installing Linux. The DOC is configured to contain two partitions, one mounted read-only which contains the firmware and crypto control software, and one mounted read/write with additional safety precautions like `noexec` and `nosuid`, for storage of configuration information and encrypted keys.

The firmware consists of a basic Linux kernel with every unnecessary service and option stripped out. This means removing support for video devices, mass storage (apart from the DOC and floppy drive), multimedia devices, and other unnecessary bagatelles. Apart from the TCP/IP stack needed by the crypto control software to communicate with the host, there are no networking components running (or even present) on the system, and even the TCP/IP stack may be absent if alternative means of communicating with the host (explained in more detail further on) are employed. All configuration tasks are performed through console access via the serial port, and software is installed by connecting a floppy drive and copying across pre-built binaries. This both minimises the size of the code base which needs to be installed on the coprocessor, and eliminates any unnecessary processes and services which might constitute a security risk. Although it would be easier if we provided a means of FTP'ing binaries across, the fact that a user must explicitly connect a floppy drive and mount it in order to change the firmware or control software makes it much harder to accidentally (or maliciously) move problematic code across to the coprocessor, provides a workaround for the fact that FTP over alternative coprocessor communications channels such as a parallel port is tricky without resorting to the use of even more potential problem software, and makes it easier to comply with the FIPS 140 requirements that (where a non-Orange Book OS is used) it not be possible for extraneous software to be loaded and run on the system. Direct console access is also used for other operations such as setting the onboard real-time clock, which is used to add timestamps to signatures. Finally, all paging is disabled, both because it isn't needed or safe to perform with the limited-write-cycle flash disk, and because it avoids any risk of sensitive data being written to backing store, eliminating a major headache which occurs with all virtual-memory operating systems [42].

At this point we have a basic system consisting of the underlying hardware and enough firmware to control it and provide the services we require. Running on top of this will be a daemon which implements the crypto control software which does the actual work.

3. Crypto Functionality Implementation

Once the hardware and functionality level of the coprocessor have been established, we need to design an appropriate programming interface for it. An interface which employs complex data structures, pointers to memory locations, callback functions, and other such elements won't work with the coprocessor unless a complex RPC mechanism is employed. Once we get to this level of complexity we run into problems both with lowered performance due to data marshalling and copying requirements and potential security problems arising from inevitable implementation bugs.

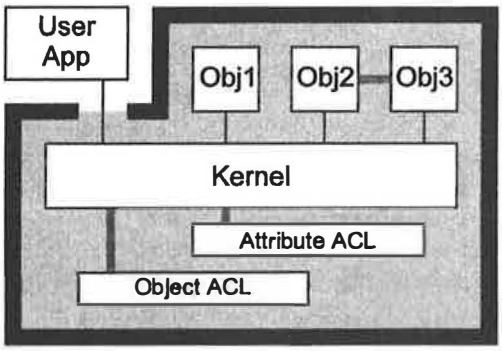


Figure 4: cryptlib architecture

A better type of interface is the one used in the cryptlib security architecture [43] which is depicted in Figure 4. cryptlib implements an object-based design which assigns unique handles to crypto-related objects but hides all further object details inside the architecture. Objects are controlled through messages sent to them under the control of a central security kernel, an interface which is ideally suited for use in a coprocessor since only the object handle (a small integer value) and

one or two arguments (either an integer value or a byte string and string length) are needed to perform most operations. This use of only basic parameter types leads to a very simple and lightweight interface, with only the integer values needing any canonicalisation (to network byte order) before being passed to the coprocessor. A coprocessor call of this type, illustrated in Figure 5, requires only a few lines of code more than what is required for a direct call to the same code on the host system. In practice the interface is further simplified by using a pre-encoded template containing all fixed parameters (for example the type of function call being performed and a parameter count), copying in any variable parameters (for example the object handle) with appropriate canonicalistion, and dispatching the result to the coprocessor. The coprocessor returns results in the same manner.

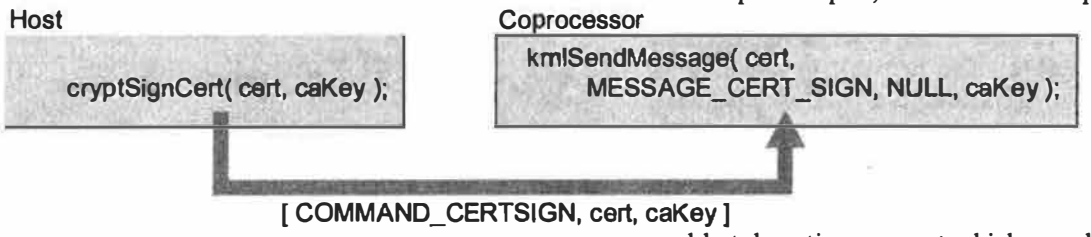


Figure 5: Communicating with the coprocessor

3.1. Communicating with the Coprocessor

The next step after designing the programming interface is to determine which type of communications channel is best suited to controlling the coprocessor. Since the embedded controller hardware is intended for interfacing to almost anything, there are a wide range of I/O capabilities available for communicating with the host. Many embedded controllers provide an ethernet interface either standard or as an option, so the most universal interface uses TCP/IP for communications. For card PC's which plug into the hosts backplane we should be able to use the system bus for communications, and if that isn't possible we can take advantage of the fact that the parallel ports on all recent PC's provide sophisticated (for what was intended as a printer port) bidirectional I/O capabilities and run a link from the parallel port on the host motherboard to the parallel port on the coprocessor. Finally, we can use more exotic I/O capabilities such as USB to communicate with the coprocessor.

The most universal coprocessor consists of a biscuit PC which communicates with the host over ethernet (or, less universally, a parallel port). One advantage which an external, removable coprocessor of this type has over one which plugs directly into the host PC is that it's very easy to unplug the entire crypto subsystem and store it separately from the host, moving it out of reach of any covert access by outsiders while the owner of the system is away. In addition to the card itself, this type of standalone setup requires a case and a power supply, either internal to the case or an external wall-wart type (these are available for about $10 with a universal input voltage range which allows them to work in any country). The same arrangement is used in a number of commercially-available products, and has the advantage that it interfaces to virtually any type of system, with the commensurate disadvantage that it requires a dedicated ethernet connection to the host (which typically means adding an extra network card), as well as adding to the clutter surrounding the machine.

The alternative option for an external coprocessor is to use the parallel port, which doesn't require a network card but does tie up a port which may be required for one of a range of other devices such as external disk drives, CD writers, and scanners which have been kludged onto this interface alongside the more obvious printers. Apart from its more obvious use, the printer port can be used either as an Enhanced Parallel Port (EPP) or as an Extended Capability Port (ECP) [44]. Both modes provide about 1-2 MB/s data throughput (depending on which vendors claims are to be believed) which compares favourably with a parallel port's standard software-intensive maximum rate of around 150 kB/s and even with the throughput of a 10Mbps ethernet interface. EPP was designed for general-purpose bidirectional communication with peripherals and handles intermixed read and write operations and block transfers without too much trouble, whereas ECP (which requires a DMA channel which can complicate the host system's configuration process) requires complex data direction negotiation and handling of DMA transfers in progress, adding a fair amount of overhead when used with peripherals which employ mixed reading and writing of small data quantities. Another disadvantage of DMA is that its use paralyses the CPU by seizing control of the bus, halting all threads which may be executing while data is being transferred. Because of this the optimal interface mechanism is EPP. From a programming point of view, this communications mechanism looks like a permanent virtual circuit which is functionally equivalent to the dumb wire which we're using the

ethernet link as, so the two can be interchanged with a minimum of coding effort.

To the user, the most transparent coprocessor would consist of some form of card PC which plugs directly into their system's backplane. Currently virtually all card PC's have ISA bus interfaces (the few which support PCI use a PCI/ISA hybrid which won't fit a standard PCI slot [45]) which unfortunately doesn't provide much flexibility in terms of communications capabilities since the only viable means of moving data to and from the coprocessor is via DMA, which requires a custom kernel-mode driver on both sides. The alternative, using the parallel port, is much simpler since most operating systems already support EPP and/or ECP data transfers, but comes at the expense of a reduced data transfer rate and the loss of use of the parallel port on the host. Currently the use of either of these options is rendered moot since the ISA card PC's assume they have full control over a passive-backplane-bus system, which means they can't be plugged into a standard PC which contains its own CPU which is also assuming that it solely controls the bus. It's possible that in the future card PC's which function as PCI bus devices will appear, but until they do it's not possible to implement the coprocessor as a plug-in card without using a custom extender card containing an ISA or PCI connector for the host side, a PC104 connector for a PC104-based CPU card, and buffer circuitry in between to isolate the two buses. This destroys the COTS nature of the hardware, limiting availability and raising costs.

The final communications option uses more exotic I/O capabilities such as USB which are present on newer embedded systems, these are much like ethernet but have the disadvantage that they are currently rather poorly supported by most operating systems.

Since we're using Linux as the resource manager for the coprocessor hardware, we can use a multithreaded implementation of the coprocessor software to handle multiple simultaneous requests from the host. After initialising the various cryptlib subsystems, the control software creates a pool of threads which wait on a mutex for commands from the host. When a command arrives, one of the threads is woken up, processes the command, and returns the result to the host. In this manner the coprocessor can have multiple requests outstanding at once, and a process running on the host won't block whenever another process has an outstanding request present on the coprocessor.

3.2. Open vs Closed-source Coprocessors

There are a number of vendors who sell various forms of tier 2 coprocessor, all of which run proprietary control software and generally go to some lengths to ensure that no outsiders can ever examine it. The usual way in which vendors of proprietary implementations try to build the same user confidence in their product as would be provided by having the source code and design information available for public scrutiny is to have it evaluated by independent labs and testing facilities, typically to the FIPS 140 standard when the product constitutes crypto hardware (the security implications of open source vs proprietary implementations have been covered exhaustively in various fora and won't be repeated here). Unfortunately this process leads to prohibitively expensive products (thousands to tens of thousands of dollars per unit) and still requires users to trust the vendor not to insert a backdoor, or accidentally void the security via a later code update or enhancement added after the evaluation is complete (strictly speaking such post-evaluation changes would void the evaluation, but vendors sometimes forget to mention this in their marketing literature). There have been numerous allegations of the former occurring [46][47][48], and occasional reports of the latter.

In contrast, an open source implementation of the crypto control software can be seen to be secure by the end user with no degree of blind trust required. The user can (if they feel so inclined) obtain the raw coprocessor hardware from the vendor of their choice in the country of their choice, compile the firmware and control software from the openly-available source code, and install it knowing that no supplemental functionality known only to a few insiders exists. For this reason the entire suite of coprocessor control software is available in source code form for anyone to examine, build, and install as they see fit.

A second, far less theoretical advantage of an open-source coprocessor is that until the crypto control code is loaded into it, it isn't a controlled cryptographic item as crypto source code and software aren't controlled in most of the world. This means that it's possible to ship the hardware and software separately to almost any destination (or source it locally) without any restrictions and then combine the two to create a controlled item once they arrive at their destination (like a two-component glue, things don't get sticky until you mix the parts).

4. Extended Security Functionality

The basic coprocessor design presented so far serves to move all security-related processing and cryptovariables out of reach of hostile software, but by taking advantage of the capabilities of the hardware and firmware used to implement it, it's possible to do much more. One of the features of the cryptlib architecture is that all operations are controlled and monitored by a central security kernel which enforces a single,

consistent security policy across the entire architecture. By tying the control of some of these operations to features of the coprocessor, it's possible to obtain an extended level of control over its operation as well as avoiding some of the problems which have traditionally plagued this type of security device.

4.1. Controlling Coprocessor Actions

The most important type of extra functionality which can be added to the coprocessor is extended failsafe control over any actions it performs. This means that instead of blindly performing any action requested by the host (purportedly on behalf of the user), it first seeks confirmation from the user that they have indeed requested that the action be taken. The most obvious application of this mechanism is for signing documents where the owner has to indicate their consent through a trusted I/O path rather than allowing a rogue application to request arbitrary numbers of signatures on arbitrary documents. This contrasts with other tier 1 and 2 processors which are typically enabled through user entry of a PIN or password, after which they are at the mercy of any commands coming from the host. Apart from the security concerns, the ability to individually control signing actions and require conscious consent from the user means that the coprocessor provides a mechanism required by a number of new digital signature laws which recognise the dangers inherent in systems which provide an automated (that is, with little control from the user) signing capability.

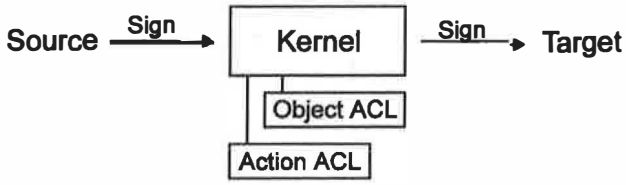


Figure 6: Normal message processing

The means of providing this service is to hook into the cryptlib kernel's sign action and decrypt action processing mechanisms. In normal processing the kernel receives the incoming message, applies various security-policy-related checks to it (for example it checks to ensure that the object's ACL allows this type of access), and then forwards the message to the intended target, as shown in Figure 6. In order to obtain additional confirmation that the action is to be taken, the coprocessor can indicate the requested action to the user and request additional confirmation before passing the message on. If the user chooses to deny the request or doesn't respond within a certain time, the request is blocked by the kernel in the same manner as if the objects ACL didn't allow it, as shown in Figure 7. This mechanism is similar to the command confirmation mechanism in the VAX A1 security kernel, which takes a command from the untrusted VMS or Ultrix-32 OS's running on top of it, requests that the user press the (non-overridable) secure attention key to communicate directly with the kernel and confirm the operation ("Something claiming to be you has requested *X*. Is this OK?"), and then returns the user back to the OS after performing the operation [49].

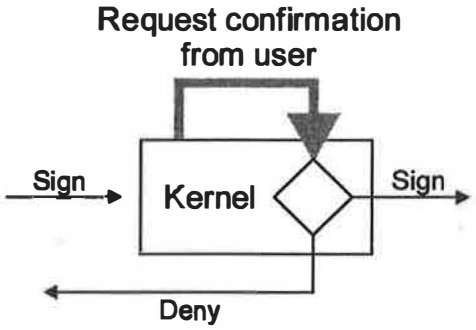


Figure 7: Processing with user confirmation

The simplest form of user interface involves two LED's and two pushbutton switches connected to a suitable port on the coprocessor (for example the parallel port or serial port status lines). An LED is activated to indicate that confirmation of a signing or decryption action is required by the coprocessor. If the user pushes the confirmation button, the request is allowed through, if they push the cancel button or don't respond within a certain time, the request is denied.

4.2. Trusted I/O Path

The basic user confirmation mechanism presented above can be generalised by taking advantage of the potential for a trusted I/O path which is provided by the coprocessor. The main use for a trusted I/O path is to allow for secure entry of a password or PIN used to enable access to keys stored in the coprocessor. Unlike typical tier 1 devices which assume the entire device is secure and use a short PIN in combination with a retry counter to protect cryptovariables, the coprocessor makes no assumptions about its security and instead relies on a user-supplied password to encrypt all cryptovariables held in persistent storage (the only time keys exist in plaintext form is when they're decrypted to volatile memory prior to use). Because of this, a simple numeric keypad used to enter a PIN isn't sufficient (unless the user enjoys memorising long strings of digits for use as passwords). Instead, the coprocessor can optionally make use of devices such as PalmPilots for password entry, perhaps in combination with novel password entry techniques such as graphical passwords [50]. Note though that, unlike a tier 0 crypto implementation, obtaining the user password via a keyboard sniffer on the host doesn't give access to

private keys since they're held on the coprocessor and can never leave it, so that even if the password is compromised by software on the host, it won't provide access to the keys.

In a slightly more extreme form, the ability to access the coprocessor via multiple I/O channels allows us to enforce strict red/black separation, with plaintext being accessed through one I/O channel, ciphertext through another, and keys through a third. Although cryptlib doesn't normally load plaintext keys (they're generated and managed internally and can never pass outside the security perimeter), when the ability to load external keys is required FIPS 140 mandates that they be loaded via a separate channel rather than over the one used for general data, which can be provided for by loading them over a separate channel such as a serial port (a number of commercial crypto coprocessors come with a serial port for this reason).

4.3. Physically Isolated Crypto

It has been said that the only truly tamperproof computer hardware is Voyager 2, since it has a considerable air gap (strictly speaking a non-air gap) which makes access to the hardware somewhat challenging (space aliens notwithstanding). We can take advantage of air-gap security in combination with cryptlib's remote-execution capability by siting the hardware performing the crypto in a safe location well away from any possible tampering. For example by running the crypto on a server in a physically secure location and tunneling data and control information to it via its built-in ssh or SSL capabilities, we obtain the benefits of physical security for the crypto without the awkwardness of having to use it from a secure location or the expense of having to use a physically secure crypto module (the implications of remote execution of crypto from a country like China with keys and crypto held in Europe or the US are left as an exercise for the reader).

Physical isolation at the macroscopic level is also possible due to the fact that cryptlib employs a separation kernel for its security [51][52], which allows different object types (and, at the most extreme level, individual objects) to be implemented in physically separate hardware. For those requiring an extreme level of isolation and security, it should be possible to implement the different object types in their own hardware, for example keyset objects (which don't require any real security since certificates contain their own tamper protection) could be implemented on the host PC, the kernel (which requires a minimum of resources) could be implemented on a cheap ARM-based plug-in card, envelope objects (which can require a fair bit of memory but very little processing power) could be implemented on a 486 card with a good quantity of memory, and encryption contexts (which can require a fair amount of CPU power but little else) could be implemented using a faster Pentium-class CPU. In practice though it's unlikely that anyone would consider this level of isolation worth the expense and effort.

5. Crypto Hardware Acceleration

So far the discussion of the coprocessor has focused on the security and functionality enhancements it provides, avoiding any mention of performance concerns. The reason for this is that for the majority of users the performance is good enough, meaning that for typical applications such as email encryption, web browsing with SSL, and remote access via ssh, the presence of the coprocessor is barely noticeable since the limiting factors on performance are set by network bandwidth, disk access times, modem speed, bloatware running on the host system, and so on. Although never intended for use as a special-purpose crypto accelerator of the type capable of performing hundreds of RSA operations per second on behalf of a heavily-loaded web server, it is possible to add extra functionality to the coprocessor through its built-in PC104 bus to extend its performance. By adding a PC104 daughterboard to the device, it's possible to enhance its functionality or add new functionality in a variety of ways, as explained below (although the prices quoted for devices will change over time, the price ratios should remain relatively constant).

5.1. Conventional Encryption/Hashing

Implementing an algorithm like DES which was originally targeted at hardware implementation, in a field-programmable gate array (FPGA) is relatively straightforward, and hash algorithms like MD5 and SHA-1 can also be implemented fairly easily in hardware by implementing a single round of the algorithm and cycling the data through it the appropriate number of times. Using a low-cost FPGA, it should be possible to build a daughterboard which performs DES and MD5/SHA-1 acceleration for around $50. Unfortunately, a number of hardware and software issues conspire to make this non-viable economically. The main problem is that although DES is faster to implement in hardware than in software, most newer algorithms are much more efficient in software (ones with large, key-dependent S-boxes are particularly difficult to implement in FPGA's because they require huge numbers of logic cells, requiring very expensive high-density FPGA's). A related problem is the fact that in many cases the CPU on the coprocessor is already capable of saturating the I/O channel (ethernet/ECP/EPP/PC104) using a pure software

implementation, so there's nothing to be gained by adding expensive external hardware (all of the software-optimised algorithms run at several MB/s whereas the I/O channel is only capable of handling around 1MB/s). The imbalance becomes even worse when any CPU faster than the entry-level 5x86/133 configuration is used, since at this point any common algorithm (even the rather slow triple DES) can be executed more quickly in software than the I/O channel can handle. Because of this it doesn't seem profitable to try to augment software-based conventional encryption or hashing capabilities with extra hardware.

5.2. Public-key Encryption

Public-key algorithms are less amenable to implementation in general-purpose CPU's than conventional encryption and hashing algorithms, so there's more scope for hardware acceleration in this area. We have two options for accelerating public-key operations, either using an ASIC from a vendor or implementing our own version with an FPGA. Bignum ASIC's are somewhat thin on the ground since the vendors who produce them usually use them in their own crypto products and don't make them available for sale to the public, however there is one company who specialise in ASIC's rather than crypto products who can supply a bignum ASIC (it's also possible to license bignum cores and implement the device yourself, this option is covered peripherally in the next section). Using this device, the PCC201 [53], it's possible to build a bignum acceleration daughterboard for around $100.

Unfortunately, the device has a number of limitations. Although impressive when it was first introduced, the maximum key size of 1024 bits and maximum throughput of 21 operations/s for 1024-bit keys and 74 operations/s for 512-bit keys compares rather poorly with software implementations on newer Pentium-class CPU's, which can achieve the same performance with a CPU speed of around 200MHz. This means that although one of these devices would serve to accelerate performance on a coprocessor based on the entry-level 5x86/133 hardware, a better way to utilise the extra expense of the daughterboard would be to buy the next level up in coprocessor hardware, giving somewhat better bignum performance and accelerating all other operations as well as a free side-effect (the entry level for Pentium-class cards is one containing a 266MHz Cyrix MediaGX, although it may be possible to put together an even cheaper one using a bare card and populating it with an AMD K6/266, currently selling for around $30). A second disadvantage of the PCC201 is that it's made available under peculiar export control terms which can make it cumbersome (or even impossible) to obtain for anyone who isn't a large company.

An alternative to using an ASIC is to implement our own bignum accelerator with an FPGA, with the advantage that we can make it as fast as required (within the limits of the available hardware). Again, there is the problem that much of the published work in the area of bignum accelerator design is by crypto hardware vendors who don't make the details available, however there is one reasonably fast implementation which achieves 83 operations/s for 1024-bit keys and 340 operations/s for 512-bit keys using a total of 6,700 FPGA basic cells (configurable logic blocks or CLB's) [54]. The use of such a large number of CLB's requires the use of very high-density FPGA's, of which the most widely-used representative is the Xilinx XC4000 family [55]. The cheapest available FPGA capable of implementing this design, the XC40200, comes with a pre-printed mortgage application form and a $2000-$2500 price tag (depending on speed grade and quantity), providing a clue as to why the design has to date only been implemented on a simulator. Again, it's possible to buy an awful lot of CPU power for the same amount of money (an equivalent level of performance to the FPGA design is obtainable using about $200 worth of AMD Athlon CPU [56]).

This illustrates a problem faced by all hardware crypto accelerator vendors, which may be stated as a derivation of Moore's law: Intel can make it faster cheaper than you can. In other words, putting a lot of effort into designing an ASIC for a crypto accelerator is a risky investment because, aside from the usual flexibility problems caused by the use of an ASIC, it'll be rendered obsolete by general-purpose CPU's within a few years. This problem is demonstrated by several products currently sold as crypto hardware accelerators which in fact act as crypto handbrakes since, when plugged in or enabled, performance slows down.

For pure acceleration purposes, the optimal price/performance tradeoff appears to be to populate a daughterboard with a collection of cheap CPU's attached to a small amount of memory and just enough glue logic to support the CPU (this approach is used by nCipher, who use a cluster of ARM CPU's in their SSL accelerators [57]). The mode of operation of this CPU farm would be for the crypto coprocessor to halt the CPU's, load the control firmware (a basic protected-mode kernel and appropriate code to implement the required bignum operation(s)) into the memory, and restart the CPU running as a special-purpose bignum engine. For x86 CPU's, there are a number of very minimal open-source protected-mode kernels which were originally designed as DOS extenders for games programming available, these ignore virtual memory,

page protection, and other issues and run the CPU as if it were very fast a 32-bit real-mode 8086. By using a processor like a K6-2 3D/333 (currently selling for around $35) which contains 32+32K of onboard cache, the control code can be loaded initially from slow, cheap external memory but will execute from cache at the full CPU speed from then on. Each of these dedicated bignum units should be capable of ~200 512-bit RSA operations per second at a cost of around $100 each.

Unfortunately the use of commodity x86 CPU's of this kind has several disadvantages. The first is that they are designed for use in systems with a certain fixed configuration (for example SDRAM, PCI and AGP busses, a 64-bit bus interface, and other high-performance options) which means that using them with a single cheap 8-bit memory chip requires a fair amount of glue logic to fake out the control signals from the external circuitry which is expected to be present. The second problem is that these CPU's consume significant amounts of power and dissipate a large amount of heat, with current drains of 10-15A and dissipations of 20-40W being common for the range of low-end processors which might be used as cheap accelerator engines. Adding more CPU's to improve performance only serves to exacerbate this problem, since the power supplies and enclosures designed for embedded controllers are completely overwhelmed by the requirements of a cluster of these CPU's. Although the low-cost processing power offered by general-purpose CPU's appears to make them ideal for this situation, the practical problems they present rules them out as a solution.

A final alternative is offered by digital signal processors (DSP's), which require virtually no external circuitry since most newer ones contain enough onboard memory to hold all data and control code, and don't expect to find sophisticated external control logic present. The fact that DSP's are optimised for embedded signal-processing tasks makes them ideal for use as bignum accelerators, since a typical configuration contains two 32-bit single-cycle multiply-accumulate (MAC) units which provide in one instruction the most common basic operation used in bignum calculations. The best DSP choice appears to be the ADSP-21160, which consumes only 2 watts and contains built-in multiprocessor support allowing up to 6 DSP's to be combined into one cluster [58]. The aggregate 3,600 MFLOPS processing power provided by one of these clusters should prove sufficient (in its integer equivalent) to accelerate bignum calculations. The feasibility of using DSP's as low-cost accelerators is currently under consideration and may be the subject of a future paper.

5.3. Other Functionality

In addition to pure acceleration purposes, it's possible to use a PC104 add-on card to handle a number of other functions. The most important of these is a hardware random number generator (RNG), since the effectiveness of the standard entropy-polling RNG using by cryptlib [59] is somewhat impaired by its use in an embedded environment. A typical RNG would take advantage of several physical randomness sources (typically thermal noise in semiconductor junctions) fed into a Schmitt trigger with the output mixed into the standard cryptlib RNG. The use of multiple independent sources ensures that even if one fails the others will still provide entropy, and feeding the RNG output into the cryptlib PRNG ensures that any possible bias is removed from the RNG output bits.

A second function which can be performed by the add-on card is to act as a more general I/O channel than the basic LED-and-pushbutton interface described earlier, providing the user with more information (perhaps via an LCD display) on what it is they're authorising.

6. Conclusion

This paper has presented a design for an inexpensive, general-purpose crypto coprocessor capable of keeping crypto keys and crypto processing operations safe even in the presence of malicious software on the host which it is controlled from. Extended security functionality is provided by taking advantage of the presence of trusted I/O channels to the coprocessor. Although sufficient for most purposes, the coprocessors processing power may be augmented through the addition of additional modules based on DSP's which should bring the performance into line with considerably more expensive commercial equivalents. Finally, the open-source nature of the design and use of COTS components means that anyone can easily reassure themselves of the security of the implementation and can obtain a coprocessor in any required location by refraining from combining the hardware and software components until they're at their final destination.

Acknowledgements

The author would like to thank Paul Karger, Sean Smith, Brian Oblivion, Jon Tidswell, Steve Weingart, Chris Zimman, and the referees for their feedback and comments on this paper.

References

[1] "Inside Windows NT", Helen Custer, Microsoft Press, 1993.

[2] "Playing Hide and Seek with Stored Keys", Nicko van Someren and Adi Shamir, 22 September 1998, presented at Financial Cryptography 1999.
[3] Eric Heimburg, "Monitoring System Events by Subclassing the Shell", *Windows Developers Journal*, **Vol.9**, **No.2** (February 1998), p.35.
[4] "Windows NT System-Call Hooking", Mark Russinovich and Bryce Cogswell, *Dr.Dobbs Journal*, January 1997, p.42.
[5] "In Memory Patching", Stone / UCF & F4CG, 1998
[6] "A *REAL* NT Rootkit, Patching the NT Kernel", Greg Hoglund, *Phrack*, **Vol.9**, **Issue 55**.
[7] "Securing Java and ActiveX", Ted Julian, Forrester Report, *Network Strategies*, **Vol.12**, **No.7** (June 1998).
[8] "Death, Taxes, and Imperfect Software: Surviving the Inevitable", Crispin Cowan and Castor Fu, *Proceedings of the ACM New Security Paradigms Workshop '98*, September 1998.
[9] "User Friendly, 6 March 1998", Illiad, 6 March 1998, `http://www.userfriendly.org/cartoons/archives/98mar/19980306.html`.
[10] "The Inevitability of Failure: The Flawed Assumption of Security in Modem Computing Environments", Peter Loscocco, Stephen Smalley, Patrick Muckelbauer, Ruth Taylor, S.Jeff Tumer, and John Farrell, *Proceedings of the 21st National Information Systems Security Conference*, October 1998.
[11] "The Importance of High Assurance Computers for Command, Control, Communications, and Intelligence Systems", W. Shockley, R. Schell, and M.Thompson, *Proceedings of the 4th Aerospace Computer Security Applications Conference*, December 1988, p.331.
[12] Jeff Schiller, quoted in *Communications of the ACM*, **Vol.42**, **No.9** (September 1999), p.10.
[13] "Software Security in an Internet World: An Executive Summary", Timothy Shimeall and John McDermott, *IEEE Software*, **Vol.16**, **No.4** (July/August 1999), p.58.
[14] "Formal Methods and Testing: Why the State-of-the-Art is Not the State-of-the-Practice", David Rosenblum, *ACM SIGSOFT Software Engineering Notes*, **Vol21**, **No.4** (July 1996), p.64.
[15] "How to bypass those pesky firewalls", Mark Jackson, in *Risks Digest*, **Vol.20**, **No.1**, 1 October 1998.
[16] "FIPS PUB 46, Data Encryption Standard", National Institute of Standards and Technology, 22 January 1988.
[17] "Federal Standard 1027, Telecommunications' General Security Requirements for Equipment Using the Data Encryption Standard", National Bureau of Standards, 14 April 1982.
[18] "FIPS PUB 46-2, Data Encryption Standard", National Institute of Standards and Technology, 30 December 1993.
[19] "Security Requirements for Cryptographic Modules", National Institute of Standards and Technology, 11 January 1994.
[20] "Building a High-Performance Programmable, Secure Coprocessor", Sean Smith and Steve Weingart, *Computer Networks and ISDN Systems*, **Issue 31** (April 1999), p.831.
[21] "Fortezza Program Overview, Version 4.0a", National Security Agency, February 1996.
[22] "iButton Home Page", `http://www.ibutton.com`.
[23] "A Tentative Approach to Constructing Tamper-Resistant Software", Masahiro Mambo, Takanori Murayama, and Eiji Okamoto, *Proceedings of the ACM New Security Paradigms Workshop '97*, September 1997.
[24] "Common Data Security Architecture", Intel Corporation, 2 May 1996.
[25] "The Giant Black Book of Computer Viruses (2nd ed)", Mark Ludwig, American Eagle Publications, 1998.
[26] "Understanding and Managing Polymorphic Viruses", Symantec Corporation, 1996.
[27] "Fravia's Page of Reverse Engineering", `http://www.fravia.org`.
[28] "Phrozen Crew Official Site", `http://www.phrozencrew.com/index2.htm`.
[29] "Stone's Webnote", `http://www.users.one.se/~stone/`.
[30] "Common Security: CDSA and CSSM, Version 2", CAE specification, The Open Group, November 1999.
[31] "Common Security Protocol (CSP)", ACP 120, 8 July 1998.
[32] "Cryptographic API's", Dieter Gollman, *Cryptography: Policy and Algorithms*, Springer-Verlag Lecture Notes in Computer Science No.1029, July 1995, p.290.
[33] "The VMEbus Handbook", VMEbus International Trade Association, 1989.
[34] "PC/104 Specification, Version 2.3", PC/104 Consortium, June 1996.
[35] "PC/104-Plus Specification, Version 1.1", PC/104 Consortium, June 1997.
[36] "EZ Dos Web Site", `http://members.aol.com/RedHtLinux/`.
[37] "The FreeDOS Project", `http://www.freedos.org`.
[38] "OpenDOS Unofficial Home Page", `http://www.deltasoft.com/opendos.htm`.
[39] "PicoBSD, the Small BSD", `http://www.freebsd.org/~picobsd/picobsd.html`.
[40] "Embedded Linux", `http://www.linuxembedded.com/`.
[41] "DiskOnChip 2000: MD2200, MD2201 Data Sheet, Rev.2.3", M-Systems Inc, May 1999.
[42] "Secure Deletion of Data from Magnetic and Solid-State Memory", Peter Gutmann, *Proceedings of the 6th Usenix Security Symposium*, July 1996.
[43] "The Design of a Cryptographic Security Architecture", Peter Gutmann, *Proceedings of the 8th Usenix Security Symposium*, August 1999.
[44] "IEEE Std.1284-1994: Standard Signaling Method for a Bi-Directional Parallel Peripheral Interface for Personal Computers", IEEE, March 1994.
[45] "PCI-ISA Passive Backplace: PICMG 1.0 R2.0", PCI Industrial Computer Manufacturers Group, 10 October 1994.

[46] "Wer ist der befugte Vierte? Geheimdienste unterwandern den Schutz von Verschlüsselungsgeräten", *Der Spiegel*, No.36, 1996, p.206.
[47] "Verschlüsselt: Der Fall Hans Buehler", Res Strehle, Werd Verlag, Zurich, 1994.
[48] "No Such Agency, Part 4: Rigging the Game", Scott Shane and Tom Bowman, *The Baltimore Sun*, 4 December 1995, p.9.
[49] "A Retrospective on the VAX VMM Security Kernel", Paul Karger, Mary Ellen Zurko, Douglas Bonin, Andrew Mason, and Clifford Kahn, *IEEE Transactions on Software Engineering*, **Vol.17, No.11** (November 1991), p1147.
[50] "The Design and Analysis of Graphical Passwords", Ian Jermyn, Alain Mayer, Fabian Monrose, Michael Reiter, and Aviel Rubin, *Proceedings of the 8th Usenix Security Symposium*, August 1999.
[51] "Design and Verification of Secure Systems", John Rushby, *ACM Operating Systems Review*, **Vol.15, No.5** (December 1981), p12.
[52] "Proof of Separability — a verification technique for a class of security kernels", John Rushby, *Proceedings of the 5th International Symposium on Programming*, Springer-Verlag Lecture Notes in Computer Science No.137 (April 1982), p.352.
[53] "Pijnenburg Product Specification: Large Number Modular Arithmetic Coprocessor, Version 1.04", Pijnenburg Custom Chips B.V., 12 March 1998.
[54] "Modular Exponentiation on Reconfigurable Hardware", Thomas Blum, MSc thesis, Worcester Polytechnic Institute, 8 April 1999.
[55] "XC4000XLA/XV Field Programmable Gate Arrays, v1.3", Xilinx, Inc, 18 October 1999.
[56] "Apache e-Commerce Solutions", Mark Cox and Geoff Thorpe, ApacheCon 2000, March 2000.
[57] nCipher, `http://www.ncipher.com`.
[58] "ADSP-21160 SHARC DSP Hardware Reference", Analog Devices Inc, November 1999.
[59] "Software Generation of Practically Strong Random Numbers", Peter Gutmann, *Proceedings of the 7th Usenix Security Symposium*, January 1998.

Secure Coprocessor Integration with Kerberos V5

Naomaru Itoi *

Center for Information Technology Integration
University of Michigan
itoi@eecs.umich.edu

Abstract

The nightmare of *Trusted Third Party (T3P)* based protocol users is compromise of the T3P. Because the compromised T3P can read and modify any user information, the entire user group becomes vulnerable to secret revelation and user impersonation. *Kerberos*, one of the most widely used network authentication protocols, is no exception. When the Kerberos *Key Distribution Center (KDC)* is compromised, all the user keys are exposed, thus revealing all the encrypted data and allowing an adversary to impersonate any user. If an adversary has physical access to the KDC host, or can obtain administrator rights, KDC compromise is possible, and catastrophic. To solve this problem, and to demonstrate the capabilities of secure hardware, we have integrated the IBM 4758 secure coprocessor into Kerberos V5 KDC. As a result of the integration, our implemented KDC preserves security even if the KDC host has been compromised.

1 Introduction

Over the past decades, numerous security protocols have been developed and have been quite successful at improving computer system security, providing safe handling of critical information. However, there still remains one large security dread; **you really do not know what your computer is doing.** Indeed, with current commodity computer technology, it is quite difficult to have confidence in system integrity[1] because (1) physical security tends to be overlooked in commodity hardware, (2) software bugs inevitably introduce security threats, and (3) new systems introduce new problems. Most of the security software available today ignores these difficulties, and simply asserts system integrity. The reasoning behind this assertion is that physical attacks are more difficult to execute than software attacks.

However, this assumption can no longer be considered reasonable as the value of information stored in computers increases. For example, the CIC Security Working Group reported the theft of a medical server that contained highly private information, such as social security numbers and medical histories of many donors at a university hospital. Considering the private nature of the stolen information, the damage the incident caused was extremely serious [19].

The time is right to begin addressing the flawed assumption that physical attack is unlikely and that system integrity is intact [32]. One approach is to employ secure hardware with the following considerations in mind. First, such hardware should be physically tamper resistant. Second, to minimize software flaws, it should be simple and throughly tested. Secure hardware is now being mass produced and is becoming more widely available (e.g., [21, 8, 43]), so it can now be more readily integrated with existing computer infrastructures. In this effort, we secure one of the most critical component in current computer systems: the trusted third party in Kerberos, namely, the *Kerberos Key Distribution Center (KDC)*.

We integrated the IBM 4758 secure coprocessor into the Kerberos KDC, which has resulted in the implemented KDC preserving critical secrets even when compromised. This paper presents the motivation, design, security consideration, implementation, and performance evaluation of the project.

We use the term "card" to refer to the 4758, and "host" to refer to the workstation to which the 4758

*This project has been carried out in the IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598, in the summer of 1999.

[1] System integrity is intact when "no unauthorized modification has been made." [10]

is attached.

1.1 Fundamental Security Problems

Physical Security

Many researchers have identified the problem of physical security of distributed computer systems [20, 49, 37]. Unlike mainframe computers of the past, in isolated computer centers, today's computer environment consists of physically distributed personal computers and workstations, connected by networks. Such an environment is difficult to protect because computers are geographically distributed, making it more difficult to control physical access. Further, PCs and workstations have weaker physical protection than mainframes in that physical access to computational and storage devices is typically possible by simply opening the cover of the computer. For example, a hard disk drive can be easily removed from a personal computer. Once it is removed, an adversary can mount it on his own computer to access it, or can make a copy and analyze it off-line. Some PCs and workstations have locks, but these tend to be of low quality and easily defeated [13].

Bootstrap Process

Arbaugh et al. have pointed out that without a secure bootstrap process, the integrity of operating system kernels cannot be trusted because malicious code (e.g., Trojan horse) can be injected in the bootstrap process [1]. For example, typical PCs can be booted from floppy disks, thus allowing arbitrary operating system kernels to run, even malicious ones or ones with Trojan horses. Some of them allow the administrator to set a BIOS password, preventing booting unless the password is entered. However, an adversary can reset the password by resetting the BIOS [13].

Software Flaws

Bugs and design flaws in software, which are unavoidable, can be exploited. For example, buffer overflow in an administrator privileged (root) process can allow an adversary to run arbitrary code with administrator privileges. Vulnerable software ranges from operating systems to applications. Some examples are as follows:

- operating systems (e.g., erroneous permission of DLL cache on Windows NT 4.0 [11])
- basic system programs (e.g., buffer overflow in `df`, `eject`, `login`, etc., in IRIX [4])
- daemons (e.g., buffer overflow in `wu-ftpd` [7], buffer overflow in IIS web server [6, 38] and bugs in sample files in IIS [48])
- applications (e.g., buffer overflow in `sendmail` [5])
- network protocols (e.g., flaws in ICMP Router Discovery Protocol allowing man-in-the-middle attack [39])
- security software (e.g., poor encryption of shell-lock [35] and Password Appraiser sending Windows passwords in the clear on the Internet [34])

Such vulnerabilities can be quite serious. For instance, they may yield administrative rights to an adversary, crash the computer system, or leak critical information.

The computer security community deals with such flaws by publishing countermeasures as soon as vulnerabilities are found. However, searching for vulnerabilities is an endless chore, as it is impossible to be confident that the software is bug-free. In addition, computer systems are developing quite rapidly, and new systems tend to bring new problems.

For example, the new functionality of Java [18] enabled client side programming on the Internet. At the same time however, a design flaw in Java caused a mismatch between the language and the bytecode, leaving the Java Virtual Machine open to attacks [29], and implementation bugs made Internet browsers vulnerable [9]. In many ways, the new technology itself enabled new kind of attacks [46, 28].

It is dangerous to assume the integrity of an operating system's kernel and software, as most software does. This is problematic especially for security critical software, such as trusted third parties.

1.2 Trusted Third Party

Several trusted third party (T3P) based security protocols are in use today. T3P is a central authority in a network that defines and enforces security policies for the other members of the network. A

certificate authority (CA) is a T3P in a public key based protocol that uses certificates for authorization. A key distribution center (KDC) is a T3P in a secret key based protocol that stores secret keys of the members. The natural match between a T3P based model and real-world security management lends T3P based protocols configurability and scalability, making them widely accepted.

The T3P is a critical point of attack on a network. The damage caused by a T3P compromise is extremely serious. In particular, it is catastrophic to have the KDC compromised, as the keys of all the members can be obtained by an adversary. With all the member keys in hand, the adversary can decrypt all the secrets encrypted with the keys, and can impersonate any member. To recover from KDC compromise, all keys must be revoked and regenerated, affecting every member. Compared with KDC, CA has better characteristics when compromised because CA stores public keys, but not private keys. However, CA compromise is still quite damaging, as an adversary can impersonate anyone by crafting bogus certificates.

Therefore, to keep systems secure, T3Ps must have the highest security. However, as described in the previous section, fundamental security problems pose a significant challenge to obtaining high levels of security with current computer systems.

1.3 IBM 4758 Secure Coprocessor

We address the problem stated above by bringing a secure coprocessor into the mix. A secure coprocessor is a "computational device that can be trusted to execute its software correctly, despite physical attack" [41].

We employ the IBM 4758 secure coprocessor because of its superior security and programmability. The 4758 is a PCI card with a tamper-resistant and tamper-responding secure coprocessor.

IBM 4758 Security

The 4758 is physically protected with layers of epoxy and metal so that it does not leak information out of the barrier, has electromagnetic shielding, and cannot be accessed without the card detecting it. The card detects opening attempts, penetration attempts, temperature attacks, and radiation attacks.

This has three types of storage: RAM, battery-backed up RAM (BBRAM), and flash memory. On detecting an attack, the card responds by resetting all the data in RAM and BBRAM, thus preventing an adversary from obtaining any information. RAM is 4 MB of volatile memory. BBRAM is 8.5 kilobytes of non-volatile secure memory. Flash is 1 MB of non-volatile memory.[2]

Validated with the FIPS 140-1 Level 4 standards, this coprocessor is one of the most trustworthy and secure coprocessors [42].

IBM 4758 Programmability

In addition to its security, the 4758 has very good programmability. Applications that run in the card are written in C, and can be debugged with a runtime debugger [12].

It has a very fast cryptographic accelerator (20 MB/s bulk DES and 20 signatures/s of RSA [3] with 1024 bit modulus), which allows for efficient implementation of security protocols. [41, 21].

It is natural to use the most secure hardware for the most critical component. To demonstrate the potential of secure hardware integration in T3P protocols, and to counter one of the fundamental security limitations of Kerberos, we integrated the 4758 into Kerberos V5 KDC.

1.4 Paper Composition

This paper presents the secure coprocessor integration with Kerberos KDC project. The next section provides the motivation behind the project by referring to related work. Section 3 describes the design of the integrated protocols. The security of our design is discussed in Section 4. Section 5 presents the prototype implementation. Performance evaluation of the prototype is presented in Section 6. Discussion and future work are in Section 7.

In this document, it is assumed that readers have some knowledge of the mechanisms of Kerberos. Readers who are not familiar with them are advised to consult available literature [3, 44, 27, 25, 26].

[2] Parameters such as storage size and cryptographic performance are reported for the 4758 Model 1. The 4758 Model 2 has improved size and performance [21].

[3] 50 MB/sec DES and 200 signatures/s RSA on 4758 Model 2

2 Related Work

This section reviews the work most closely related to our research. Section 2.1 introduces Kerberos. Section 2.2 describes approaches taken by researchers to address goals similar to ours, and the relationship between their approaches and ours. Section 2.3 discusses secure hardware integration with the Kerberos client.

2.1 Kerberos

Kerberos [44, 27, 26] is a very widely used authentication protocol. It is a secret key, T3P protocol based on the Needham-Schroeder protocol [36]. Kerberos KDC offers two services, *Authentication Service (AS)*, and *Ticket Granting Service (TGS)*. AS authenticates members (*principals*), while TGS establishes a session key between two principals. For example, Alice needs to run AS with the Kerberos KDC to prove she is Alice, and needs to run TGS with the Kerberos KDC to obtain a ticket, which is later sent to Bob to establish a session key between them. Every principal in the protocol, i.e., users, services [4], and computers, is assigned a secret key, which is shared between the principal and the KDC.

Kerberos is used in universities to protect their computer network environments. CMU, Cornell, MIT, Stanford, the University of Michigan, and many more embrace it. It is also part of the products of many corporations such as Transarc, Cisco, Qualcomm, IBM, and Microsoft, whose Windows 2000 employs it as a fundamental network authentication method. Many network applications are modified to work with Kerberos, including login, ftp, telnet, PAM, ssh, AFS, and DFS. Its security has been thoroughly analyzed [2, 27], and it scales quite well. For example, three replicated Kerberos servers at the University of Michigan serve 180,000 users. It is also quite portable. Both the clients and the servers run on almost any UNIX or Windows systems. We believe that Kerberos will continue to be an important security system.

A huge security issue for Kerberos is the common problem of KDCs, that is, it yields all the keys when compromised [2, 32]. MIT Kerberos V5-1.0.6 stores a master key, which encrypts the other keys, in cleartext in a file (/usr/local/var/krb5kdc/.k5.DOMAINNAME). Keys of the principals, i.e., the user keys and the service keys, are encrypted with the master key and are stored on hard disks. Therefore, if an adversary has administrative rights for the KDC's computer or physical access to its disks, all the keys can be stolen.

[4] Examples of services are login service and ftp service.

2.2 Public Key Based Authentication Systems

Several public key authentication systems have been designed and implemented [47, 40, 45] that are compatible with or related to Kerberos. Many of these systems are similar to ours in the sense that they try to protect the trusted third party. The logic to support them is that public key based authentication systems fail more gracefully than secret key based systems when T3P is compromised. Indeed, CA does not store private keys, thus maintaining forward secrecy and preventing an adversary from getting immediate impersonation ability. However, these systems amplify the value of our work because of the following:

- Even in public key systems, the trusted third party (CA) is the most critical point of attack. By obtaining the CA's private key, an adversary can modify certificates, issue bogus certificates, and modify certificate revocation lists, to impersonate members. Therefore, it is vital to protect the CA with secure hardware.
- Because of both the computational overhead of public key cryptography and the necessity for key revocation, we have concerns over how public key based authentication systems scale. In contrast, we know the secret key based system scales quite well. We believe that secret key based Kerberos will be in service for a long time.

Therefore, we believe that public key augmentation to Kerberos complements our work.

2.3 Smartcard Integration with Kerberos Client

Smartcard integration with the Kerberos client enhances security of Kerberos by taking advantage of secure hardware in the form of smartcard [24, 33, 15, 31, 22]. This work (smartcard/Kerberos client) and our work (secure coprocessor/Kerberos server) complement each other.

3 Design

As described in Section 1.1, we prefer not to trust the host computer on which the Kerberos KDC runs. Thus, we designed our protocol to survive a host "hijack". One way of achieving this is to implement an entire KDC in the 4758, but we did not take this approach, as this will limit the performance and scalability of KDC. Instead, we decided to split a KDC between the host and the 4758, following these design policies.

- keys never leave the 4758 in clear
- all cryptographic operations are executed in the 4758

More concretely, the master key is stored in the battery-backed up RAM in the 4758 and never leaves. Because of storage limitations, user keys are stored in the host and encrypted with the master key. The 4758 has BBRAM of 8.5 kilobytes and 1 megabytes of flash memory, allowing it to securely store many DES keys. However, storage in the host is more abundant than storage in the 4758, and a Kerberos realm, for example, at a university, may require a huge number of keys, so we decided to store them in the host.

When user keys are used, for example, to encrypt a ticket, they are downloaded from the host to the 4758, decrypted there with the master key, used, and then deleted from its memory. Session keys are also generated in the 4758, augmented into tickets, and encrypted in the 4758 before being shipped to clients.

3.1 process_as_req

Figure 1 shows how the authentication request (AS_REQ) is handled in Kerberos V5. The keys (the user keys of Alice and Bob, the `krbtgt` key, and the master key) are used in the host. The master key is not shown in the figure, but is used to decrypt the other keys. If the host is compromised, all the keys are revealed.

To solve this problem, we designed the protocol with the 4758 in Figure 2. Note that all the encryption and decryption is done in the 4758, and no key is in the host in the clear. This protects the keys from an adversary who compromises the host.

The 4758 generates the ticket and the reply only if requests are consistent, namely, the following conditions are met:

- Key of Alice is used in preauthentication
- Alice is the client name in the ticket
- Bob is the server name in the ticket
- Key of Bob is used to encrypt the ticket
- Bob is the server name in the reply
- Key of Alice is used to encrypt the reply

Otherwise, it rejects the requests. As a result, the adversary cannot fool the 4758 to generate tickets and replies for her advantage.

3.2 process_tgs_req

Figures 3 and 4 show ticket granting service request (TGS_REQ) handling both with and without the 4758.

In the protocol using the 4758, all the encryption is done in the 4758, and no key is in the host in the clear. Consistency checks similar to the ones in `process_as_req` take place.

4 Security Analysis

In this section, we discuss the security of the design presented in Section 3.

4.1 Model

We start with constructing a model of our system. The model consists of the following participants:

Alice (A) A Kerberos principal who uses the authentication and ticket granting service provided by KDC. Alice's workstation is assumed to be trustworthy. This allows us to combine Alice and her workstation into one object.

Bob (B) A Kerberos principal with which Alice wants to establish mutual authentication. Bob's workstation is assumed to be trustworthy.

KDC-host Software component of KDC that resides on a host computer.

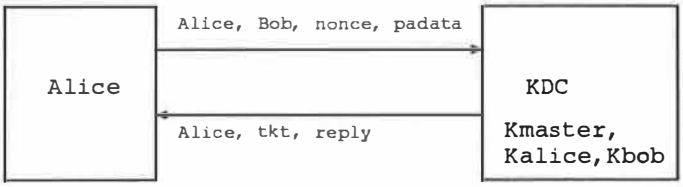


Figure 1: original AS_REQ processing in Kerberos V5

Alice	The principal who wants to be authenticated.
Bob	The principal with whom Alice wants to communicate. (Bob is the "krbtgt" when AS_REQ is used to obtain TGT.)
PAdata	{current time}Kalice : Preauthentication data to prove that Alice knows the right Kalice.
Kses	Session key
Tkt	{Alice, Bob, Kses}Kbob : Ticket forwarded by Alice to Bob to prove that Alice carried out authentication with the KDC. If Bob is "krbtgt", the tkt is the TGT ({Alice, krbtgt, Kses}Kkrbtgt)
Reply	{Bob, nonce, Kses}Kalice : Alice decrypts it to get the session key.

The parties send additional information, such as message types, protocol version number, flags, and start/expiration/renew-until time. We omit them in this figure because they are treated the same with or without the 4758.

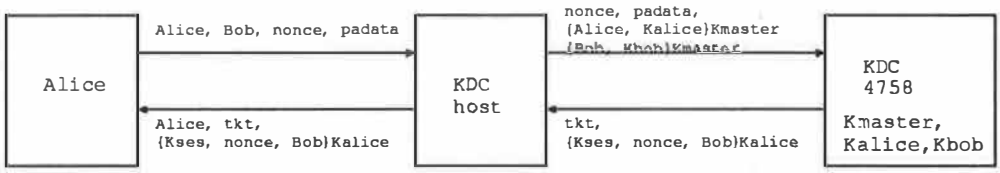


Figure 2: AS_REQ processing in Kerberos V5 with the 4758

Security critical tasks, e.g., en(de)cryption and random key generation, are moved from the host to the 4758. The host sends the 4758 the information needed for such tasks, e.g., the nonce sent by Alice, and the encrypted keys of Alice and Bob. The 4758 generates a reply to Alice, and sends it back to the host.

The entries encrypted with the master key e.g., {Alice, Kalice}Kmaster, are generated and decrypted in the 4758. The KDC host stores them (encrypted) in the Kerberos database and sends them to the 4758 when needed. The KDC host does not know the master key.

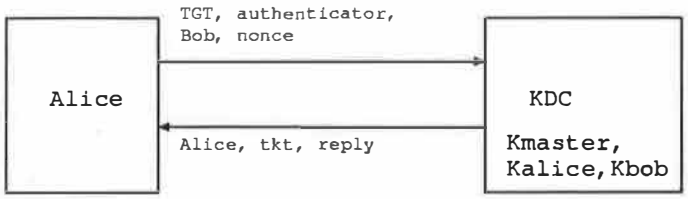


Figure 3: original TGS_REQ processing in Kerberos V5

Alice	The principal who wants to be authenticated.
Bob	The principal with whom Alice wants to communicate. Bob is "krbtgt" when TGS_REQ is used to obtain TGT.
PAdata	Preauthentication data (TGT and Authenticator). and knows the right Kses.
TGT	{Alice, krbtgt, Kses}Kkrbtgt : Ticket Granting Ticket, which proves that Alice carried out authentication with KDC.
Authenticator	{Alice, time, (subkey)}Kses
K	Key in TGT or subkey in authenticator
Kses	Session key
Tkt	{**Alice, Bob, Kses'**}K : New ticket for Alice and Bob.
Reply	{**Bob, nonce, Kses'**}K : Alice decrypts it to get the session key.

The parties send additional information, such as message types, protocol version number, flags, and start/expiration/renew-until time. We omit them in this figure because they are treated the same with or without the 4758.

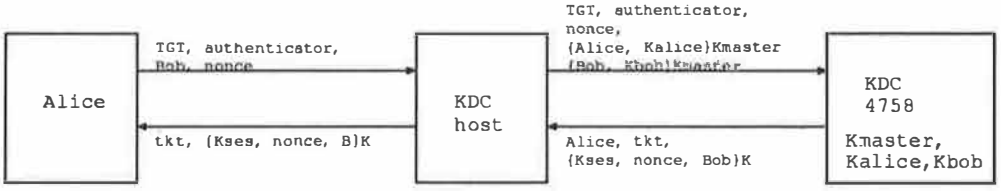


Figure 4: TGS_REQ processing in Kerberos V5 with the 4758

Security critical tasks, e.g., en(de)cryption and random key generation, are moved from the host to the 4758. The host sends the 4758 the information needed for such tasks, e.g., the nonce sent by Alice, the encrypted keys of Alice and Bob. The 4758 generates a reply to Alice, and sends it back to the host.

The entries encrypted with the master key, e.g., {**Alice, Kalice**}**Kmaster**, are generated and decrypted in the 4758. KDC host stores them (encrypted) in the Kerberos database and sends them to the 4758 when needed. The KDC host does not know the master key.

KDC-4758 Software component of KDC that resides on a secure coprocessor.

Mallory (M) An adversary.

4.2 Threats

We make the following assumptions in our model. Some of these assumptions are discussed in detail in a related paper [22].

1. System administration is appropriately done.

   As problems of system administration are out of this paper's scope, administration is assumed to be done appropriately, namely, (1) the master key is stored in KDC-4758 and nowhere else, and (2) keys of Alice and Bob are stored in KDC-host encrypted with the master key. We discuss more about administration in Section 7.3.

2. Client workstations are secure.

   As problems of the security of client workstations are out of this paper's scope, client workstations are assumed to be secure, namely, (1) a client workstation does not steal user's information, and (2) it does not alter or modify messages a user sends.

3. Passwords of Alice and Bob are good.

   The problem of dictionary attack against user chosen passwords is out of this paper's scope; passwords are assumed to be chosen carefully so that the dictionary attack against them is impossible.

4. DES is strong.

   Our principal cipher is DES, which is assumed impossible to compromise in reasonable amount of time. This may not be a good assumption any more in the age of fast DES crackers [14], but Kerberos will eventually replace DES with triple-DES, thus eliminating the brute force attack to DES.

5. Mallory can compromise KDC-host.

   Mallory can read and modify any information in KDC-host, and can make KDC-host do anything she wants.

6. Mallory cannot compromise KDC-4758.

   Mallory can neither read nor modify any information in KDC-4758. When she tries, 4758 deletes all the information in it. Mallory cannot influence the behavior of KDC-4758.

7. Mallory can read, modify, and alter messages in the network connecting the participants.

8. Mallory can be a principal in the Kerberos realm.

4.3 Attacks

4.3.1 Key Theft

Without 4758

Mallory can steal all keys by compromising KDC-host. This is possible by Assumption 5.

With 4758

Mallory cannot steal any key. The master key is in KDC-4758, and is not readable (Assumption 1, 6). All the other keys are in KDC-host, but are encrypted by the master key, with DES, which is unbreakable (Assumption 4).

4.3.2 User Impersonation

Without 4758

Mallory can impersonate any user by stealing or guessing the user key.

With 4758

Mallory cannot impersonate any user. First, she cannot steal a user key. Second, the other way of impersonating a user (Alice) is to obtain a ticket $\{\text{Alice, Bob, } K_{A,B}\}K_B$ and the session key $K_{A,B}$. Mallory can obtain the ticket by sniffing the network (Assumption 7), but this is impossible as well. The session key is generated in KDC-4758 and is always encrypted by K_A or K_B when it is outside KDC-4758. K_A and K_B are strong (Assumption 3), so the session key cannot be obtained. These keys cannot be stolen from client workstations (Assumption 2).

4.3.3 Ticket / Reply Forgery

Without 4758

Mallory can generate any ticket or reply by using stolen keys.

With 4758

Mallory cannot generate a ticket or reply on her advantage. KDC-4758 generates them only after Alice showes possession of her key through preauthentication, and consistency is checked as described in Section 3.1.

5 Implementation

We implemented the AS and TGS protocols described in Section 3 by modifying Kerberos V5-1.0.6 distributed by MIT. The host platform is Linux 2.0.36 (RedHat 5.2) on an IBM PC. The secure coprocessor is the IBM 4758 Model 1, with the Secure Cryptographic Coprocessor toolkit version 1.33.

5.1 Outline

The implementation was carried out in the following three steps.

- analysis of `process_as_req()` and `process_tgs_req()`, which implement AS and TGS to identify which portions of the functions should be moved to the 4758
- implementation of the card side functions that have functionality equivalent to the portions identified in the first step
- modification of the host side program to make calls to the implemented functions in the card

5.2 Step1: Functionality Analysis

There are six parts to be moved in AS: three calls to key decryption and one each to preauthentication, ticket encryption, and reply encryption. Likewise, there are six parts in TGS: two calls to key decryption and one each to ticket decryption, authenticator decryption, ticket encryption, and reply encryption.

As the performance evaluation in Section 6 shows, the overhead of calling a function in the 4758 is high. Therefore, to obtain high performance, the six calls should be combined into one call. However, as cryptographic code and non-cryptographic code are tightly coupled together in Kerberos, doing so changes the order of execution and breaks modularity, thus significantly complicating the host side code. For this prototype, we decided to make six calls in each AS and TGS, valuing simplicity and manageability over performance. A detailed look at the overhead in Section 6.2 explains our decision.

5.3 Step2: Card Side Functions

5.3.1 Authentication Service

Key Decryption

User keys are stored in the host and encrypted with the master key. The card decrypts the keys before using them. The host-side `decrypt_key()` function sends keys to the card, decrypts them and then stores them in RAM for future use. The function is called three times in AS: first for Alice's key for preauthentication, second for Bob's key for ticket encryption, and third for Alice's key for reply encryption. [5]

Preauthentication

Preauthentication is the step in which Alice proves her identity to the KDC by proving knowledge of her key. By default, preauthentication takes place in the following three steps:

- Alice sends to the KDC a timestamp encrypted with her key : {time}Kalice.
- The KDC decrypts {time}Kalice.
- The KDC verifies that the request is really generated by Alice by determining whether the time value falls within clock skew allowed. KDC goes on to the next step in AS if the answer is yes. Otherwise, KDC rejects Alice.

Because this step requires the use of Alice's user key, this function is moved to the card. The 4758 decrypts the timestamp and verifies it. If the timestamp is invalid, following requests, e.g., ticket encryption and reply encryption, are rejected.

Ticket Encryption

A ticket is a data structure sent from the KDC to Alice to establish a session key. Roughly speaking, it is

[5] We can save one call by caching the key in preauthentication and using it in reply encryption. We did not try this optimization for the prototype; performance is not yet our goal.

{Alice, Bob, Kses}Kbob. Part of the ticket is not security critical, and is generated in the host. Afterward, the ticket is sent to the card, filled with the session key generated in the card, encrypted with Kbob, and sent to Alice. The card stores the session key for future use because the reply will include it as described in the next paragraph.

Reply Encryption

Similar to the ticket, the reply {Bob, nonce, Kses}Kalice includes a public part, which is encoded in the host and is sent to the card. The session key, generated in the ticket encryption function, is inserted into the reply. The card then encrypts the reply with Alice's key.

5.3.2 Ticket Granting Service

As in AS, six calls are made to the card in TGS: two calls to key decryption, and one each to ticket decryption, authenticator decryption, ticket encryption, and reply encryption. Some of the functions are common in AS; we explain only the functions that do not appear in AS.

Ticket Decryption

TGS decrypts the ticket granting ticket, or TGT ({Alice, krbtgt, Kses}Kkrbtgt), to obtain Alice's name and the session key. Because it involves the TGS key (Kkrbtgt), and the session key is in the TGT, this step must be carried out in the card. The card decrypts the TGT and returns it in the clear to the host excluding the session key. The session key must not leave the card, so it is stored in RAM in the card. Later it is used in authenticator decryption and reply encryption.

Authenticator Decryption

The authenticator {Alice, time, (subkey)}Kses is decrypted in the card. The timestamp is checked in the card.

5.4 Step 3: Host Side Modification with Secure Hardware RPC

As with other secure hardware [23], the communication methods between the host and the 4758 are primitive. For example, the only type provided is an array of bytes. It is the developers' responsibility to convert types such as `int`, `short`, `char`, and more complicated data structures, into and out of the string of bytes. It is a burden for developers to deal with low-level programming, e.g., marshaling and unmarshaling data structures, dealing with endian problems, message buffer handling, and error handling.

To provide a better abstraction, we developed the *Secure Hardware Remote Procedure Call (SHRPC)*, which parses the interface definition file and generates C programs to handle the low-level communication details. With SHRPC, procedure call abstraction is provided to the host. As a consequence, modification in the host side is merely to call SHRPC stub functions, e.g., `decrypt_key()`, instead of equivalent but more elaborate functions in the host.

Although *interface definition language (IDL)* should follow some standards, such as `rpcgen`, we picked our own simple IDL for rapid implementation. The interface definition file for the Kerberos / 4758 integration looks like this:

```
# Interface Declaration File
#  for the Kerberos V5 / 4758 Project
# 8/6/1999, Naomaru Itoi
PROG:   krb5_4758
FUNC:   decrypt_key
IN:
int type
# type :
#   0: client key
#   1: server key
string enc_key
OUT:
int tick
...
```

6 Performance Evaluation

We evaluated the performance of the prototype in the following environment: IBM Netfinity PC with Intel 300 MHz Pentium; the IBM 4758 secure coprocessor model 1; the KDC and the Kerberos clients running on the same computer to avoid network delay.

Each measurement was carried out ten times and an average is presented in tables. Variance was very small.

6.1 Overall Result

This section describes the performance of AS. `kinit` is the client program used to initiate the AS request. The total time kinit spends with or without 4758 is shown. To exclude the time spent for password typing, the password is hard coded in the `kinit` program. `kinit` with the 4758 takes 34% more time than `kinit` without it.

	time (sec)
`kinit` without 4758	0.0611
`kinit` with 4758	0.0820

`sclient` is the client we used to initiate the TGS request. `sclient` with the 4758 takes 33% more time than `sclient` without it.

	time (sec)
`sclient` without 4758	0.0719
`sclient` with 4758	0.0953

4758 integration introduces approximately 33% of overhead in both cases. We look into the details in the following sections.

6.2 Communication Overhead

In this section, we examine communication overhead. We measure the total time spent for the six cryptographic operations described in Section 5.2, the time spent in the card, and derive the communication overhead. As shown in Figure 5, the total time is the sum of the card time and the overhead.

	Total	Card Time	Overhead
AS w/o 4758	0.00054	-	-
AS w/ 4758	0.02535	0.00866	0.01669
TGS w/o 4758	0.00032	-	-
TGS w/ 4758	0.02748	0.00866	0.01882

Communication overhead is approximately twice as much as the card time in both AS and TGS. This is an obvious bottleneck and there is an obvious optimization. Theoretically, the number of calls can be reduced from six in each AS and TGS to one in AS and two in TGS. All operations can be done at once in AS. In TGS, the TGT (ticket granting ticket) must be decrypted to obtain the name of the client before the KDC looks up its database. In contrast, ticket encryption and reply encryption must happen after the database lookup. Therefore, TGS requires two calls. This optimization would reduce the card time to 0.00278 seconds in AS and 0.00314 seconds in TGS, thus reducing the overhead of using 4758 to 11% in AS and 15% in TGS.

6.3 Card Time Details

Although communication overhead was the bottleneck, it is also useful to study the details of the time spent in the card. Breakdown of AS and TGS is shown in the following table. For each function, total time and time spent in main components are presented.

AS

function	contents	time (sec)
decrypt_key	24B DES decryption	0.000957
	TOTAL	0.001109
kdc_preauth	40B DES decryption	0.000997
	CPGetTime	0.000023
	TOTAL	0.001445
encrypt_tk	168B DES encryption	0.001191
	random number gen	0.000352
	random number gen	0.000352
	168B CRC	0.000041
	TOTAL	0.002078
encode_kdc	216B DES encryption	0.001270
	random number gen	0.000352
	216B CRC	0.000053
	TOTAL	0.001809

TGS

function	contents	time (sec)
decrypt_key	24B DES decryption	0.000957
	TOTAL	0.001115
decrypt_tk	168B DES decryption	0.001172
	168B CRC	0.000041
	TOTAL	0.001324
rd_rec_dec	120B DES decryption	0.001113
	120B CRC	0.000031
	TOTAL	0.001230
encrypt_tk	168B DES encryption	0.001191
	random number gen	0.000352
	random number gen	0.000352
	168B CRC	0.000041
	TOTAL	0.002105
encode_kdc	184B DES encryption	0.001211
	random number gen	0.000352
	184B CRC	0.000045
	TOTAL	0.001773

DES operation takes the longest time. Considering that the hardware implemented DES takes much longer time than the software implemented CRC even though the hardware itself is quite fast (20 MB/s [6]), we believe the most of the DES opera-

[6] 50 MB/s on Model 2.

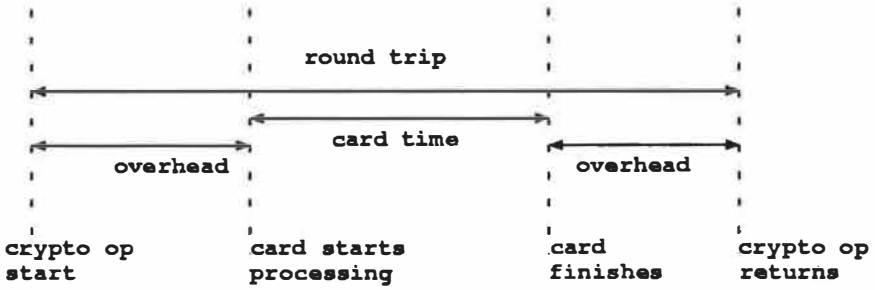


Figure 5: total, card time, and overhead

tion time is spent in making a system call to DES hardware and setting up a key schedule. For an application that operates on such small data (100 - 200 bytes), which we believe many authentication and authorization systems do, it is good to provide (1) software implementation of crypto operations to save system call overhead and (2) a decoupled API for DES key scheduling separate from DES operation. (2) is helpful because some keys are used repeatedly, e.g., a master key.

7 Discussion

7.1 Implementation Limitations

Due to time limitation, our implementation has the following limitations.

User Name - Key Binding

In Sections 3 and 4, discussions were made assuming a user name and the key of the user are encrypted together. However, this is not the case in our prototype because the key data structure in MIT Kerberos V5-1.* does not include the user name in it.

Preauthentication Failure

When preauthentication fails, either because it is not encrypted with an appropriate key, or timestamps do not match, the 4758 should reject the following operations, namely, ticket encryption and reply operations. This has not been implemented yet.

Consistency Check

Consistency check described in Section 3.1 has not been implemented.

TGS Authenticator Check

Authenticator check described in Section 5.3.2 has not been implemented yet. The 4758 simply decrypts and returns the authenticator.

7.2 Lessons Learned

Integration of secure hardware into a security protocol can be significantly simplified if the original implementers of the protocol anticipate the use of secure hardware. Complication of our work comes from cryptographic operations and non-cryptographic operations being tightly coupled in a program, e.g., they coexist in one function. If they are decoupled cleanly in an initial implementation, the work of integration is merely to move the crypto code to the secure hardware. Moreover, we believe the separation is good for portability of the protocol, e.g., to switch from one encryption system to another.

7.3 Future Work

Several steps must be taken before this project is deployed.

Complete Implementation

Unfinished implementation, discussed in Section 7.1 should be completed to realize the claimed security.

Administration

We have not addressed problems associated with administration: changing passwords, adding / removing principals, changing the the KDC's policy, etc.

The *Kerberos Database* (KDB) is the database in which Kerberos stores its critical information, e.g., the keys and the principal attributes; it is accessed by administrators through `kadmind`. Because the data in the KDB are sensitive, the entries are encrypted with the master key. As a consequence, in the 4758 integrated KDC, administration requests must go through the 4758.

An adversary can attack a Kerberos/4758 system by attacking the channel between the administrator and the 4758. For example, one possible attack, which could reduce the advantage of integrating the 4758 into Kerberos, is a Trojan horse in the administrator's terminal. If it can interrupt the operations by the administrator, it can steal sensitive information, e.g., user passwords. In fact, this secure I/O problem is a general concern for any security system, which requires the administrator be trustworthy, and the administrator's terminal be secure.

We plan to address these concerns by carrying out mutual authentication and establishing an encrypted connection between a system administrator and the 4758 with Kerberos authentication, and using the connection to securely transfer requests by the administrator to the 4758.

This will partially achieve our goals because the administrator is authenticated via Kerberos, and communication is encrypted. However, it is not possible to provide a completely trusted terminal with current commercial hardware, even with secure hardware such as the 4758, because even if the processors and storage are trusted, the I/O devices may not be. For example, a keyboard or a display instrumented with a hardware eavesdropper can steal administrators' keystrokes. However, it is much easier to keep a terminal secure during administration than to keep a Kerberos server secure in 24 hours a day, seven days a week fashion. Therefore, we defer solving this problem of secure I/O.

Performance Optimization

As described in Section 6.2, the six calls to the 4758 in AS and TGS should be combined into one and two calls respectively to optimize the performance. The drawback of this optimization is that it changes the Kerberos code significantly. In the Kerberos/Cartel meeting in July of 1999, we sensed that such a radical change would pose a major challenge to Kerberos developers with regard to maintaining the source code. Therefore, we decided to first implement a prototype to determine what the computer systems community thinks about it before proceeding to the deployment step.

Brute Force Attack to Master Key

If an adversary has access to messages passed between the host and the 4758, he or she can obtain a plaintext-ciphertext pair. Some messages are encrypted with single DES and the master key. This is problematic because given a plaintext-ciphertext pair, DES key can be cracked by a brute force attack in a week [14]. Kerberos distribution from MIT supports triple DES, eliminating this threat.

Replay Attack

An adversary can use a replay attack to impersonate Alice if he or she hijacks the host and has Alice's obsolete password. Here we describe a possible attack and the countermeasure.

Our Kerberos/4758 protocol stores the master key inside the 4758, which encrypts the other keys with this master key and stores the ciphertext on the host. An adversary (Mallory) cannot access these plaintext keys even if she compromises the host because she does not know the master key, which never leaves the 4758.

However, without additional measures, such a protocol suffers from replay attacks if Mallory can learn one of Alice's old passwords. The replay attack is carried out as follows:

- Mallory obtains an old password of Alice, Pa.
- Because Mallory has complete access to the host, she can obtain {Alice, Kalice}Kmaster.
- Alice, knowing that her password is stolen, changes her password to Pa'. At this point, the old password Pa and the corresponding key Kalice are obsolete.
- Mallory types the obsolete password, Pa. Pa is hashed to the key Kalice. The KDC hijacked by Mallory sends {Alice, Kalice}Kmaster to the 4758. If the 4758 does not know Kalice is obsolete, it thinks Kalice is fresh, and sends a reply

encrypted by Kalice to the KDC/Mallory. Mallory successfully decrypts the reply, thus impersonating Alice.

To avoid this attack, we use key version numbering and obsolete key caching. First, all the keys in the Kerberos database have a key version number, N. This key version number is different from the key version number used in the original Kerberos V5 protocol. An encrypted key entry contains this version number, i.e., {Alice, Kalice, N}. When Alice changes her password, Alice's current key version number is updated to N+1. The 4758 generates a new key entry {Alice, Kalice', N+1}, sends the entry back to the host, and caches a pair {Alice, N+1} in its internal memory.

The 4758 checks the cache whenever it receives a key from the host. If the version numbers do not match, then the key received is obsolete. To avoid cache overflow, once in a while (e.g., daily) the 4758 regenerates the new N and computes the new entries for all the keys, and sends them back to the host.

The cache should not overflow too quickly. If the cache size is 1MB and each entry is 32 bytes, then the maximum number of entries in the cache is 32K entries — which we imagine exceeds the maximum number of password changes in a day. (Furthermore, some preliminary tests indicate that the time needed for cryptographic operations to regenerate the cache is not excessive.)

8 Conclusion

This paper demonstrates the ability of secure hardware to improve the security of current computer environments. We can shrink the security boundary of the trusted third party from a workstation to a secure coprocessor, which is a smaller and more secure component. The implemented Kerberos KDC survives host compromise.

We plan to make the source code freely available.

9 Acknowledgement

I thank Sean Smith, Ronald Perez, and Dawn Song at IBM research for their advice and discussion, especially for bringing up the replay attack described in Section 7.3, and suggesting the countermeasures.

I thank Peter Honeyman at the University of Michigan, Mark Lindemann and Joan Dyer in IBM research, and the Kerberos developers for their help and discussion. I thank Chris Feak and Evan Cordes at the University of Michigan for assistance on English writing.

References

[1] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[2] S. M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. In *Proceedings of the Winter 1991 Usenix Conference*, January 1991. ftp://research.att.com/dist/internet_security/kerblimit.usenix.ps.

[3] Bill Bryant. Designing an authentication system: a dialogue in four scenes, 1997. Afterword by Theodore Ts'o. http: //web.mit.edu/kerberos/ www/ dialogue.html.

[4] CERT/CC. Cert advisory ca-97.21 (sgi buffer overflow vulnerabilities), July 1997. http:// www.cert.org/ advisories/ CA-97.21.sgi_buffer_overflow.html.

[5] CERT/CC. Cert advisory ca-97.05 (mime conversion buffer overflow in sendmail versions 8.8.3 and 8.8.4), January 1999. http:// www.cert.org/ advisories/ CA-97.05.sendmail.html.

[6] CERT/CC. Cert advisory ca-99-07 (iis buffer overflow), June 1999. http:// www.cert.org/ advisories/ CA-99-07-IIS-Buffer-Overflow.html.

[7] CERT/CC. Cert advisory ca-99-13 (multiple vulnerabilities in wu-ftpd), Oct 1999. http:// www.cert.org/ advisories/ CA-99-13-wuftpd.html.

[8] Chrysalis-its. http:// www.chrysalis-its.com/.

[9] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From hotjava to netscape and beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996. http:// www.cs.princeton.edu/ sip/ pub/secure96.html.

[10] Dorothy Denning. *Cryptography and Data Security.* Addison-Wesley, 1983.

[11] dildog@l0pht.com. any local user can gain administator privileges and/or take full control over the system. L0pht Security Advisory, February 1999. http:// www.l0pht.com/ advisories.html.

[12] J. Dyer, R. Perez, S.W. Smith, and M. Lindemann. Application support architecture for a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference*, October 1999.

[13] Kevin Fenzi and Dave Wreski. Linux security howto, April 1999.

[14] Electronic Frontier Foundation. *Cracking DES - Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly & Associates, Inc., 1 edition, 1998.

[15] G. Gaskell, J. Trinkle, and M. Warner. Smart card integration with kerberos. In *16th Australian Computer Science Conference*, pages 479 – 485, Melbourne, Australia, February 1996.

[16] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications. In *Proceedings of 6th USENIX Unix Security Symposium*, July 1996.

[17] Li Gong. Java security: Present and near future. IEEE Micro, May/June 1997.

[18] James Gosling and Henry McGilton. The java language environment. White Paper, May 1996. http:// java.sun.com/ docs/ white/ langenv/.

[19] CIC Security Working Group. *Final Report: Incident Cost Analysis and Modeling Project.* Committee on Institutional Cooperation, 1997.

[20] Maurice Herlihy and Doug Tygar. How to make replicated data secure. In *Proceedings of CRYPTO-87*, pages 379 – 391, 1988.

[21] IBM. Cryptographic cards home page. http:// www.ibm.com/ security/ cryptocards.

[22] Naomaru Itoi and Peter Honeyman. Smartcard integration with Kerberos V5. In *Proceedings of USENIX Workshop on Smartcard Technology*, Chicago, May 1999.

[23] Naomaru Itoi, Peter Honeyman, and Jim Rees. Scfs: A unix filesystem for smartcards. In *Proceedings of USENIX Workshop on Smartcard Technology*, Chicago, May 1999. To appear.

[24] M. Krawjewski Jr. Concept for a smart card kerberos. In *National Computer Security Conference*, number 15, pages 76 – 83. National Institute of Science and Technology, US Department of Commerce, 1992.

[25] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security.* Prentice Hall, 1995.

[26] John T. Kohl and B. Clifford Neuman. The Kerberos network authentication service (V5), September 1993. Request For Comments 1510.

[27] John T. Kohl, B. Clifford Neuman, and Theodore Y. T'so. *Distributed Open Systems*, chapter The Evolution of the Kerberos Authentication System, pages 78–94. IEEE Computer Society Press, 1994.

[28] Mark D. Ladue. Hostile applets home page. http:// metro.to/ mladue/ hostile-applets/.

[29] Mark D. Ladue. When java was one: Threats from hostile byte code. In *Proceedings of the 20th National Information Systems Security Conference*, 1997.

[30] Jay Lepreau, Brian Ford, and Mike Hibler. The persistent relevance of the local operating system to global applications. In *Proceedings of the 7th ACM SIGOPS European Workshop*, September 1996.

[31] Mark Looi, Paul Ashley, Loo Tang Seet, Richard Au, Gary Gaskell, and Mark Vandenwauver. Enhancing SEMAME V4 with smart cards. In *CARDIS'98*, Louvain-la-Neuve, Belgium, Sept. 1998. Third Smart Card Research and Advanced Application Conference.

[32] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *21st National Information Systems Security Conference*, Crystal City, Virginia, October 1998. National Security Agency, NISSC. http://www.jya.com / paperF1.htm.

[33] Jr. Marjan Krajewski, John C. Chipchak, David A. Chodorow, and Jonathan T Trostle. Applicability of smart cards to network user authentication. *The USENIX Association, Computing Systems*, 7(1):75 – 89, 1994.

[34] mudge@l0pht.com. Users of the tool password apraiser are unwittingly publishing nt user passwords to the internet. L0pht Security Advisory, January 1999. http:// www.l0pht.com/ advisories.html.

[35] mudge@l0pht.com and lumpy. Users can de-obfuscate and retrieve the hidden shell code. L0pht Security Advisory, October 1999. http:// www.l0pht.com/ advisories.html.

[36] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993 – 999, December 1978.

[37] Innovative Security Products. Computer security. White Paper, 1998.

[38] Ryan R Permeh (rrpermeh@RCONNECT.COM). Re: Retina vs. iis4, round 2, ko, June 1999. http:// www.rootshell.com/.

[39] sili@l0pht.com. Attackers can remotely add default route entries. L0pht Security Advisory, August 1999. http:// www.l0pht.com/ advisories.html.

[40] M. Sirbu and J. Chuang. Distributed authentication in kerberos using public key cryptography. In *Internet Society 1997 Symposium on Network and Distributed System Security*, 1997.

[41] S. W. Smith and S. H. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks, (Special Issue on Computer Network Security)*, 31:831 – 860, April 1999.

[42] S.W. Smith, R. Perez, S.H. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference*, October 1999.

[43] Spyrus. http:// www.spyrus.com/.

[44] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX Conference*. USENIX, February 1988.

[45] Brian Tung, Matthew Hur, Ari Medvinsky, Sasha Medvinsky, John Wray, and Jonathan Trostle. Public key cryptography for cross-realm authentication in kerberos. IETF Internet Draft, expires December 1, 1999. draft-ietf-cat-kerberos-pk-cross-09.txt.

[46] J. D. Tygar and Alma Whitten. Www electronic commerce and java trojan horses. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, November 1996.

[47] M. Vandenwauver, R. Govaerts, and J. Vandewalle. Overview of authentication protocols: Kerberos and sesame. In *Proceedings 31st Annual IEEE Carnahan Conference on Security Technology*, pages 108 – 113, 1997.

[48] weld@l0pht.com. Web users can view sensitive information in microsoft iis 4.0 web server. L0pht Security Advisory, May 1999. http:// www.l0pht.com/ advisories.html.

[49] Bennet Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, May 1994.

Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor

John Scott Robin
U.S. Air Force
scott_robin_@hotmail.com

Cynthia E. Irvine *
Naval Postgraduate School
irvine@cs.nps.navy.mil
http://cisr.nps.navy.mil

Abstract

A virtual machine monitor (VMM) allows multiple operating systems to run concurrently on virtual machines (VMs) on a single hardware platform. Each VM can be treated as an independent operating system platform. A secure VMM would enforce an overarching security policy on its VMs.

The potential benefits of a secure VMM for PCs include: a more secure environment, familiar COTS operating systems and applications, and enormous savings resulting from the elimination of the need for separate platforms when both high assurance policy enforcement, and COTS software are required.

This paper addresses the problem of implementing secure VMMs on the Intel Pentium architecture. The requirements for various types of VMMs are reviewed. We report an analysis of the virtualizability of all of the approximately 250 instructions of the Intel Pentium platform and address its ability to support a VMM. Current "virtualization" techniques for the Intel Pentium architecture are examined and several security problems are identified. An approach to providing a virtualizable hardware base for a highly secure VMM is discussed.

1 Introduction

A virtual machine monitor (VMM) is software for a computer system that creates efficient, isolated programming environments that are "duplicates" which provide users with the appearance of direct access to the real machine environment. These duplicates are referred to as virtual machines. Goldberg [12] defines a virtual machine (VM) as: "a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in native mode". A VMM manages the real resources of the computer system, exporting them to virtual machines.

A VMM offers a number of benefits not found in conventional multiprogramming systems.

1.1 VMM Benefits

First, virtual machine monitors normally allow a system manager to **configure the environment** in which a VM will run. VM configurations can be different from those of the real machine. For example, a real machine might have 32MB of memory, but a virtual machine might have only 8 MB. This would allow a developer to test an application on a machine with only 8 MB of memory without having to construct a hardware version of that real machine.

Second, virtual machine monitors allow **concurrent execution of different operating systems** on the same hardware. Users can run any operating system and applications designed to run on the real processor architecture. Thus application development for different operating systems is easier. A developer can easily test applications on many operating systems simultaneously while running on the same base platform.

Third, virtual machine monitors allow users to **isolate untrusted applications of unknown quality**. For example, a program downloaded from the Internet could be tested in a VM. If the program contained a virus, the virus would be isolated to that VM. A secure VMM will ensure that other high integrity VMs and their applications and data are protected from corruption.

Fourth, virtual machine monitors can be used to **upgrade** operating system software to a different version without losing the ability to run the older "legacy" operating system and its applications. The legacy software can run in a virtual machine exactly as it did previously on the real machine, while the new version of the operating system runs in a separate virtual machine.

Finally, VMMs can be used to construct **system software for scalable computers** that have anywhere from 10 to 100 processors. VMMs can facilitate the develop-

*The opinions in this paper are those of the authors and should not be construed to reflect those of their employers.

ment of functional and reliable system software for these computers.

Using a VMM, an additional software layer can be inserted between the hardware and multiple operating systems. This VMM layer would allow multiple copies of an operating system to run on the same scalable computer. The VMM also allows these operating systems to share resources with each other. This solution has most of the features of an operating system custom-built for a scalable machine, but with lower development costs and reduced complexity. Disco, developed for the Stanford FLASH shared-memory multi-processor [8] is an example of this solution. It uses different commercial operating systems to provide high-performance system software.

1.2 VMM Characteristics and Layers

A VMM has three characteristics [28]. First, a VMM provides **an execution environment almost identical to the original machine**; any program executed on a VM should run the same as it would on an unvirtualized machine. Exceptions to this rule result from differences in system resource availability, timing dependencies, and attached I/O devices. If resource availability, e.g. physical memory, is different, the program will perform differently. Timing dependencies may lose their validity because a VMM may intervene and execute a different set of instructions when certain instructions are executed by a VM. Finally, if the VM is not configured with all of the peripheral devices required by the real machine, application behavior will differ.

Second, a VMM must be **in control of real system resources**. No program running on a VMM can access any resource not explicitly allocated to it by the VMM. Also the VMM can regain control of previously allocated resources.

Efficiency is the third VMM characteristic. A large percentage of the virtual processor's instructions must be executed by the machine's real processor, without VMM intervention. Instructions which cannot be executed directly by the real processor are interpreted by the VMM. Some virtual machines exhibit the recursion property: it is possible to run a VMM inside of a VM, producing a new level of virtual machines. The real machine is normally called Level 0. A VMM running on Level 0 is said to be Level 1, etc.

1.3 VMM Logical Modules

A VMM normally has three generic modules: dispatcher, allocator, and interpreter. A jump to the dispatcher is placed in every location to which the machine traps. The dispatcher then decides which of its modules to call when a trap occurs. The second type of module is the allocator. If a VM tries to execute a privileged instruction that would change the resources of the VM's environment, the VM will trap to the VMM dispatcher. The dispatcher will handle the trap by invoking the allocator that performs the requested resource allocation according to VMM policy. A VMM has only one allocator module, however, it accounts for most of the complexity of the VMM. It decides which system resources to provide to each VM, ensuring that two different VM's do not get the same resource. The final module type is the interpreter. For each privileged instruction, the dispatcher will call an interpreter module to simulate the effect of that instruction. This prevents VMs from seeing the actual state of the real hardware. Instead they see only their virtual machine state.

1.4 Attractions of a Secure VMM

An isolated VM constrained by an overarching security policy enforced by the underlying secure VMM is attractive. Also, VMM technology provides stronger isolation of virtual machines than found in conventional multiprogramming environments [21]. Within a constrained VM, legacy operating systems and applications are executed unmodified and are easily upgraded and replaced even within the context of rapidly evolving software product lifecycles.

In the past, some virtual machine monitors, such as the SDC KVM/370 [11, 9, 33, 10] and the DEC VAX SVS [17], have been used to separate mandatory security classes. A secure VMM for the Intel Pentium[1] processor architecture would be very desirable because a single machine could be used to implement critical security policies while also running popular Win32 operating systems and applications.

Although the x86 processor family has been used as the base for many highly secure systems [23, 27, 26, 25, 24], it has not been considered as a VMM base. Recent increased interest in VMM technology suggests that a popular hardware base for a new generation of VMMs would be highly attractive. Before embarking on such a venture, its feasibility must be carefully examined. This paper presents an analysis to determine whether the Intel Pentium architecture can support a highly secure VMM without sacrificing user convenience.

[1] Throughout this paper, the term "Intel Pentium architecture" will refer to the architecture of the following processors, which are all trademarks of the Intel Corporation: Intel Pentium, Intel Pentium Pro, Intel Pentium with MMX Technology, Intel Pentium II and Intel Pentium III.

1.5 Paper Organization

The rest of this paper is organized as follows: Section 2 discusses the three different types of VMMs and their hardware requirements. Section 3 is an analysis of the Intel Pentium architecture with respect to the VMM hardware requirements described in Section 2. Section 4 asks if a VMM designed for the Intel Pentium architecture can be secure. Finally, Section 5 presents our conclusions and future research.

2 VMM Requirements

This section discusses each type of VMM including the Type I VMM, Type II VMM, and Hybrid VMM. It will also cover the architectural features required for the successful implementation for each VMM type.

2.1 Virtual Machine Monitors Types

An operating system consists of instructions to be executed on a hardware processor. When an operating system is virtualized, some portion, ranging from none to all, of the instructions may be executed by underlying software. The amount of software and hardware execution of processor instructions determines if one has a complete software interpreter machine (CSIM), hybrid VM (HVM), VMM, or a real machine. Each of these different types of machines provides a normal machine environment, meaning that processor instructions can be executed on them, viz. a VMM can host an operating system. However, they differ in the way that the machine environment actually executes the processor instructions. A real machine uses only direct execution: the processor executes every instruction of the program directly. A CSIM uses only software interpretation: a software program emulates every processor instruction. There has been a recent resurgence of interest in CSIM architectures [3, 18]. A VMM requires that a "statistically dominant subset" of the virtual processor's instructions be executed on the real processor [12]. Performance will be effected by the size of the subset.

VMMs primarily use direct execution, with occasional traps to software. As a result, the performance of VMMs is better than CSIMs and HVMs. An HVM is a VMM that uses software interpretation on all privileged instructions. HVMs are possible on a larger class of systems than VMMs. The definition of a VMM does not specify how the VMM gains control of the machine to interpret instructions that cannot be directly executed on the processor. As a result, there are two different types of VMMs that can create a virtual machine environment. These types are referred as Type I and Type II [12]. A Type I VMM runs on a bare machine. It is an operating system with virtualization mechanisms. It performs the scheduling and allocation of the system's resources. A Type II VMM runs as an application. The operating system that controls the real hardware of the machine is called the "host OS." The host OS does not need or use any part of the virtualization environment. Every OS that is run in the Type II virtual environment is called a "guest OS." In a Type II VMM, the host operating system provides resource allocation and a standard execution environment to each guest OS.

2.2 Execution of Privileged Instructions

When executing in a virtual machine, some processor instructions can not be executed directly on the processor. These instructions would interfere with the state of the underlying VMM or host OS and are called sensitive instructions. The key to implementing a VMM is to prevent the direct execution of sensitive instructions. Some sensitive instructions in the Intel Pentium architecture are privileged, meaning that if they are not executed at most privileged hardware domain, they will cause a general protection exception. Normally, a VMM is executed in privileged mode and a VM is run in user mode; when privileged instructions are executed in a VM, they cause a trap to the VMM. If all sensitive instructions of a processor are privileged, the processor is considered to be "virtualizable:" then, when executed in user mode, all sensitive instructions will trap to the VMM. After trapping, the VMM will execute code to emulate the proper behavior of the privileged instruction for the virtual machine. However, if sensitive, non-privileged instructions exist, it may be necessary for the VMM to examine all instructions before execution to force a trap to the VMM when a sensitive, non-privileged instruction is encountered.

The most severe performance penalty occurs when running a complete software interpreter machine (CSIM) on the same hardware. A CSIM emulates every instruction of the real processor. It does not meet Goldberg's definition [12] of a virtual machine because it does not execute any of the instructions directly on the real processor.

The following sections summarize Goldberg's analysis of processor requirements for the types of VMMs he identified.

2.3 Type I VMM

A Type I VMM runs directly on the machine hardware. It is an operating system or kernel that has mechanisms to support virtual machines. It must perform scheduling and resource allocation for all virtual machines in the system and requires drivers for hardware peripherals.

To support a Type I VMM, a processor must meet three virtualization requirements:

Requirement 1 The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode. For example, a processor cannot use an additional bit in an instruction word or in the address portion of an instruction when in privileged mode.

Requirement 2 There must be a method such as a protection system or an address translation system to protect the real system and any other VMs from the active VM.

Requirement 3 There must be a way to automatically signal the VMM when a VM attempts to execute a sensitive instruction. It must also be possible for the VMM to simulate the effect of the instruction.

Sensitive instructions include:

Requirement 3A Instructions that attempt to change or reference the mode of the VM or the state of the machine.

Requirement 3B Instructions that read or change sensitive registers and/or memory locations such as a clock register and interrupt registers.

Requirement 3C Instructions that reference the storage protection system, memory system, or address relocation system. This class includes instructions that would allow the VM to access any location not in its virtual memory.

Requirement 3D All I/O instructions.

2.4 Type II VMM

A Type II VMM runs as an application on a host operating system and relies on the host OS for memory management, processor scheduling, resource allocation, and hardware drivers. It provides only virtualization support services. To support a Type II virtual machine a processor must meet all of the hardware requirements for the Type I VMM listed above. In addition, the following software requirements must be met by the host operating system of the Type II VMM:

Weaker Requirement 3A The *host OS* cannot do anything to invalidate the requirement that the method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode.

Requirement 2 Primitives must be available in the *host OS* to protect the VMM and other VMs from the active virtual machine. Examples include a protection primitive, address translation primitive, or a sub-process primitive.

When the virtual machine traps because it has attempted to execute a sensitive instruction, the host OS must direct the signal to the VMM. Therefore, the host OS needs a primitive to perform this action. The host OS also needs a mechanism to allow a VMM to run the virtual machine as a sub-process. The VMM must be able to simulate sensitive instructions.

A highly secure Type II VMM will require a highly secure host OS because it will depend upon the host OS. Flaws in the host OS would undermine the security of the Type II VMM.

2.5 Hybrid VMM

Often, if a processor does not meet the Type I or Type II VMM requirements, it can still implement a hybrid virtual machine monitor (HVM). A hybrid VMM has all of the advantages of normal VMMs and avoids the performance penalties of a CSIM. It is functionally equivalent to the real machine. However, an HVM and a VMM differ in that an HVM interprets every privileged instruction in software, whereas a VMM may directly execute some privileged instructions. An HVM treats the privileged mode of hardware as a pure software construct. In both a VMM and an HVM, all non-privileged instructions execute directly on the processor.

An HVM has less strict hardware requirements than a VMM in tow ways. First, the HVM does not have to directly execute non-sensitive privileged instructions because they are all emulated in software. Second, because of the emulation, the HVM need not provide additional mapping of the the most privileged processor mode into another processor privilege level. However, increased interpretative execution usually lowers the performance of an HVM relative to a VMM.

The hardware requirements for an HVM result in the following changes to the original Type I VMM requirements. First, Requirement 1, which states that the method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode, is eliminated. Second, Requirement 3A, which states that if an instruction attempts to change or reference the mode of the VM or the state of the machine, there must be a way to simulate the instruction, is weakened.

3 Pentium Architecture and VMMs

Goldberg [12] identified the key architectural features of third generation hardware pertinent to virtual machines:

- two processor modes of operation,
- a method for non-privileged programs to call privileged system routines,
- a memory relocation or protection mechanism such as segmentation or paging, and

- asynchronous interrupts to allow the I/O system to communicate with the CPU.

All of these still apply to the Intel Pentium architecture. It has four modes of operation, known as rings, or current privilege level (CPL), 0 through 3. Ring 0, the most privileged, is occupied by operating systems. Application programs execute in Ring 3, the least privileged. The Pentium also has a method to control transfer of program execution between privilege levels so that non-privileged tasks can call privileged system routines: the *call gate*. The Pentium also uses both paging and segmentation to implement its protection mechanisms. Finally, the Pentium uses both interrupts and exceptions to allow the I/O system to communicate with the CPU. The architecture has 16 predefined interrupts and exceptions and 224 user-defined, or maskable, interrupts.

Despite these features, the ability of the Pentium architecture to support virtualization is likely to be serendipitous as the processor was not explicitly designed to support virtualization. This section reports an analysis of the virtualizability of the Pentium against the hardware requirements described in Section 2. Every documented instruction for the Intel Pentium [2] was analyzed for its ability to support virtualization [30].

Any instruction in the processor's instruction set that violates rule 1, 2, 3 (3A, 3B, 3C, or 3D) will preclude the processor from running a Type I or Type II VMM. Additionally, any instruction that violates rule 2, 3A in its weaker form, 3B, 3C, or 3D prevents the processor from running an HVM. By combining these two statements, one can see that any instruction that violates rule 2, 3A in its weaker form, 3B, 3C, or 3D makes the processor non-virtualizable.

With respect to the VMM hardware requirements listed above, Intel meets all three of the main requirements for virtualization.

Requirement 1: *The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode.* Intel meets this requirement because the method for executing privileged and non-privileged instructions is the same. The only difference between the two types of instructions in the Intel architecture is that privileged instructions cause a general protection exception if the CPL is not equal to 0.

Requirement 2:*There must be a method such as a protection system or an address translation system to protect the real system and any other VMs from the active VM.* Intel uses both segmentation and paging to implement its protection mechanism. Segmentation provides a mechanism to divide the linear address space into individually protected address spaces (segments). Segments have a *descriptor privilege level* (DPL) ranging from 0 to 3 that specifies the privilege level of the segment. The DPL is used to control access to the segment. Using DPLs, the processor can enforce boundaries between segments to control whether one program can read from or write into another program's segments.

Requirement 3:*There must be a way to automatically signal the VMM when a VM attempts to execute a sensitive instruction. It must also be possible for the VMM to simulate the effect of the instruction.* The Intel architecture uses interrupts and traps to redirect program execution and allow interrupt and exception handlers to execute when a privileged instruction is executed by an unprivileged task. However, the Pentium instruction set contains **sensitive, unprivileged instructions**. The processor will execute unprivileged, sensitive instructions without generating an interrupt or exception. Thus, a VMM will never have the opportunity to simulate the effect of the instruction.

After examining each member of the Pentium instruction set, it was found that seventeen instructions violate Requirement 3. All seventeen instructions violate either part B or part C of Requirement 3 and make the Intel processor non-virtualizable. To construct a truly virtualizable Pentium chip one must focus on these instructions. They are discussed in more detail below.

3.1 Sensitive Register Instructions

Several Intel instructions break hardware virtualization Requirement 3B. The rule states that instructions are sensitive if they read or change sensitive registers and/or memory locations such as a clock register and interrupt registers.

3.1.1 SGDT, SIDT, and SLDT Instructions

The SGDT, SIDT, and SLDT instructions violate this rule in a similar way. In protected mode, all memory accesses pass through either the *global descriptor table* (GDT) or *local descriptor table* (LDT). The GDT and LDT contain segment descriptors that provide the base address, access rights, type, length, and usage information for each segment. The interrupt descriptor table (IDT) is similar to the GDT and LDT, but it holds gate descriptors that provide access to interrupt and exception handlers. The GDTR, LDTR, and IDTR all contain the linear addresses and sizes of their respective tables.

All three of these instructions (SGDT, SIDT, SLDT) store a special register value into some location. The SGDT instruction stores the contents of the GDTR in a 6-byte memory location. The SLDT instruction stores the segment selector from the LDTR in a 16 or 32-bit general-purpose register or memory location. The SIDT

[2] The analysis was based on available documentation as of 22 June 1999 and involved approximately 250 instructions.

Table 1: **Important CR0 Machine Status Word Bits**

Bit	Flag Name	Description
0	PE - Protection Enable	Enable Protected Mode when set and real mode when clear
1	MP - Monitor Coprocessor	Controls the interaction of the WAIT or FWAIT instruction with the TS flag.
2	EM - Emulation	If clear, processor has an internal or external floating point unit
3	TS - Task Switched	Allows delayed saving of the floating point unit context on a task switch until the unit is accessed by the new task.
4	ET - Extension Type	For 386 and 468 processors, indicates whether an Intel 387 DX math co-processor is present (hard-coded to 1 on >Pentium processors).
5	NE - Numeric Error	Enables internal or PC-style mechanism for FPU error reporting.

instruction stores the contents of the IDTR in a 6-byte memory location. These instructions are normally only used by operating systems but are not privileged in the Intel architecture. Since the Intel processor only has one LDTR, IDTR, and GDTR, a problem arises when multiple operating systems try to use the same registers. Although these instructions do not protect the sensitive registers from reading by unprivileged software, the processor allows partial protection for these registers by only allowing tasks at CPL 0 to load the registers. This means that if a VM tries to write to one of these registers, a trap will be generated. The trap allows a VMM to produce the expected result for the VM. However, if an OS in a VM uses SGDT, SLDT, or SIDT to reference the contents of the IDTR, LDTR, or GDTR, the register contents that are applicable to the host OS or Type I VMM will be given. This could cause a problem if an operating system of a virtual machine (VMOS) tries to use these values for its own operations: it might see the state of a different VMOS executing within a VM running on the same VMM. Therefore, a Type I VMM or Type II VMM must provide each VM with its own virtual set of IDTR, LDTR, and GDTR registers.

3.1.2 SMSW Instruction

The SMSW instruction stores the machine status word (bits 0 through 15 of control register 0) into a general-purpose register or memory location. Bits 6 through 15 of CR0 are reserved bits that are not supposed to be modified. Bits 0 through 5, however, contain system flags that control the operating mode and state of the processor and are described in Table 1.

Although this instruction only stores the machine status word, it is sensitive and unprivileged. Consider the following scenario: A VMOS is running in real mode within the virtual environment created by a VMM running in protected mode. If the VMOS checked the MSW to see if it was in real mode, it would incorrectly see that the PE bit is set. This means that the machine is in protected mode. If the VMOS halts or shuts down if in protected mode, it will not be able to run successfully.

This instruction is only provided for backwards compatibility with the Intel 286 processor [16]. Programs written for the Intel 386 processor and later are supposed to use the MOV instruction to load and store control registers, which are privileged instructions. Therefore, SMSW could be removed and only systems requiring backward compatibility with the Intel 286 processor would be affected. Application software written for the Intel 286 and 8086 processors should be unaffected because the SMSW instruction is a system instruction that should not be used by application software.

3.1.3 PUSHF and POPF Instructions

The PUSHF and POPF instructions reverse each other's operation. The PUSHF instruction pushes the lower 16 bits of the EFLAGS register onto the stack and decrements the stack pointer by 2. The POPF instruction pops a word from the top of the stack, increments the stack pointer by 2, and stores the value in the lower 16 bits of the EFLAGS register. The PUSHFD and POPFD instructions are the 32-bit counter-parts of the POPF and PUSHF instructions. Pushing the EFLAGS register onto the stack allows the contents of the EFLAGS register to be examined. Much like the lower 16 bits of the CR0 register, the EFLAGS register contains flags that control the operating mode and state of the processor. Therefore, the PUSHF/PUSHFD instructions prevent the Intel processor from being virtualizable in the same way that the SMSW instruction prevents virtualization. In virtual-8086 mode, the IOPL must equal 3 to use the PUSHF instructions. Of the 32 flags in the EFLAGS register, fourteen are reserved and six are arithmetic flags. Table 2 describes the bits of concern.

The POPF instruction allows values in the EFLAGS register to be changed. Its varies based on the processor's current operating mode. In real-mode, or when operating at CPL 0, all non-reserved flags in the EFLAGS

Table 2: **Important EFLAGS Register Bits**

Bit	Flag Name	Description
8	TF - Trap	Set to enable single-step mode for debugging.
9	IF - Interrupt Enable	Controls processor response to maskable interrupt requests.
10	DF - Direction	If set, string instructions process addresses from high to low.
12-13	IOPL - I/O Privilege Level	I/O privilege level of the currently running task.
14	NT - Nested Task	Set when the current task is linked to the previous task.
16	RF - Resume	Controls processor response to debug exceptions.
17	VM - Virtual-8086 Mode	Enables Virtual-8086 mode when set.
18	AC - Alignment Check	Enables alignment checking of memory references.
19	VIF - Virtual Interrupt	Virtual image of the IF flag.
20	VIP - Virtual Interrupt Pending	Indicates whether or not an interrupt is pending.
21	ID - Identification	If a program can set or clear this instruction, the CPUID instruction is supported.

register can be modified except the VM, VIP, and VIF flags. In virtual-8086 mode, the IOPL must equal 3 to use the POPF instructions. The IOPL allows an OS to set the privilege level needed to perform I/O. In virtual-8086 mode, the VM, RF, IOPL, VIP, and VIF flags are unaffected by the POPF instruction. In protected mode, there are several conditions based on privilege levels. First, if the CPL is greater than 0 and less than or equal to the IOPL, all flags can be modified except IOPL, VIP, VIF, and VM. The interrupt flag is altered when the CPL is at least as privileged as the IOPL. Finally, if a POPF/POPFD instruction is executed without enough privilege, an exception is not generated. However, the bits of the EFLAGS register are not changed.

The POPF/POPFD instructions also prevent processor virtualization because they allow modification of certain bits in the EFLAGS register that control the operating mode and state of the processor.

3.2 Protection System References

Many Intel instructions violate Requirement 3C: Instructions are sensitive if they reference the storage protection system, memory or address relocation system.

3.2.1 LAR, LSL, VERR, VERW

Four instructions violate the rule in a similar manner: LAR, LSL, VERR, and VERW. The LAR instruction loads access rights from a segment descriptor into a general purpose register. The LSL instruction loads the unscrambled segment limit from the segment descriptor into a general-purpose register. The VERR and VERW instructions verify whether a code or data segment is readable or writable from the current privilege level. The problem with all four of these instructions is that they all perform the following check during their execution: (CPL $\rightarrow$ DPL) OR (RPL $\rightarrow$ DPL). This conditional checks to ensure that the current privilege level (located in bits 0 and 1 of the CS register and the SS register) and the requested privilege level (bits 0 and 1 of any segment selector) are both greater than the descriptor privilege level (the privilege level of a segment). This is a problem because a VM normally does not execute at the highest privilege (i.e., CPL = 0). It is normally executed at the user or application level (CPL = 3) so that all privileged instructions will cause traps that can be handled by the VMM. However, most operating systems assume that they are operating at the highest privilege level and that they can access any segment descriptor. Therefore, if a VMOS running at CPL = 3 uses any of the four instructions listed above to examine a segment descriptor with a DPL < 3, it is likely that the instruction will not execute properly.

3.2.2 POP Instruction

The reason that the POP instruction prevents virtualization is very similar to that mentioned in the previous paragraph. The POP instruction loads a value from the top of the stack to a general-purpose register, memory location, or segment register. However, the POP instruction cannot be used to load the CS register since it contains the CPL. A value that is loaded into a segment register must be a valid segment selector. The reason that POP prevents virtualization is because it depends on the value of the CPL. If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL, a general protection exception is raised. Additionally, if the DS, ES, FS, or GS register is being loaded, the segment being pointed to is a nonconforming code segment or data, and the RPL and CPL are greater than the DPL, a general protection exception is raised. As in the previous case, if a VM's CPL

is 3, these privilege level checks could cause unexpected results for a VMOS that assumes it is in CPL 0.

3.2.3 PUSH Instruction

The PUSH instruction also prevents virtualization because it references the protection system. The PUSH instruction allows a general-purpose register, memory location, an immediate value, or a segment register to be pushed onto the stack. This cannot be allowed because bits 0 and 1 of the CS and SS register contain the CPL of the current executing task. The following scenario demonstrates why these instructions could cause problems for virtualization. A process that thinks it is running in CPL 0 pushes the CS register to the stack. It then examines the contents of the CS register on the stack to check its CPL. Upon finding that its CPL is not 0, the process may halt.

3.2.4 CALL, JMP, INT n, and RET

The CALL instruction saves procedure linking information to the stack and branches to the procedure given in its destination operand. There are four types of procedure calls: near calls, far calls to the same privilege level, far calls to a different privilege level, and task switches. Near calls and far calls to the same privilege level are not a problem for virtualization. Task switches and far calls to different privilege levels are problems because they involve the CPL, DPL, and RPL. If a far call is executed to a different privilege level, the code segment for the procedure being accessed has to be accessed through a call gate. A task uses a different stack for every privilege level. Therefore, when a far call is made to another privilege level, the processor switches to a stack corresponding to the new privilege level of the called procedure. A task switch operates in a manner similar to a call gate. The main difference is that the target operand of the call instruction specifies the segment selector of a task gate instead of a call gate. Both call gates and task gates have many privilege level checks that compare the CPL and RPL to DPLs. Since the VM normally operates at user level (CPL 3), these checks will not work correctly when a VMOS tries to access call gates or task gates at CPL 0.

The discussion above on LAR, LSL, VERR, and VERW provides a specific example of how running a CPL 0 operating system as a CPL 3 task could cause a problem. The JMP instruction is similar to the CALL instruction in both the way that it executes and the reasons it prevents virtualization. The main difference between the CALL and the JMP instruction is that the JMP instruction transfers program control to another location in the instruction stream and does not record return information.

The INT instruction is also similar to the CALL instruction. The INT n instruction performs a call to the interrupt or exception handler specified by n. INT n does the same thing as a far call made using the CALL instruction except that it pushes the EFLAGS register onto the stack before pushing the return address. The INT instruction references the protection system many times during its execution.

The RET instruction has the opposite effect of the CALL instruction. It transfers program control to a return address that is placed on the stack (normally by a CALL instruction). The RET instruction can be used for three different types of returns: near, far, and inter-privilege-level returns. Much like the CALL instruction, the inter-privilege-level far return examines the privilege levels and access rights of the code and stack segments that are being returned to determine if the operation should be allowed. The DS, ES, FS, and GS segment registers are cleared by the RET instruction if they refer to segments that cannot be accessed by the new privilege level. Therefore, RET prevents virtualization because having a CPL of 3 (the VM's privilege level) could cause the DS, ES, FS, and GS registers to not be cleared when they should be. The IRET/IRETD instruction is similar to the RET instruction. The main difference is it returns control from an exception, interrupt handler, or nested task. It prevents virtualization in the same way that the RET instruction does.

3.2.5 STR Instruction

Another instruction that references the protection system is the STR instruction. The STR instruction stores the segment selector from the task register into a general-purpose register or memory location. The segment selector that is stored with this instruction points to the task state segment of the currently executing task. This instruction prevents virtualization because it allows a task to examine its requested privilege level (RPL). Every segment selector contains an index into the GDT or LDT, a table indicator, and an RPL. The RPL is represented by bits 0 and 1 of the segment selector. The RPL is an override privilege level that is checked (along with the CPL) to determine if a task can access a segment. The RPL is used to ensure that privileged code cannot access a segment on behalf of an application unless the application also has the privilege to access the segment. This is a problem because a VM does not execute at the highest CPL or RPL (RPL = 0), but at RPL = 3. However, most operating systems assume that they are operating at the highest privilege level and that they can access any segment descriptor. Therefore, if a VM running at a CPL and RPL of 3 uses STR to store the contents of the task register and then examines the infor-

mation, it will find that it is not running at the privilege level at which it expects to run.

3.2.6 MOVE Instruction

Two variants of the MOVE instruction prevent Intel processor virtualization. These are the two MOV instructions that load and store control registers. The MOV opcode that stores segment registers allows all six of the segment registers to be stored to either a general-purpose register or to a memory location. This is a problem because the CS and SS registers both contain the CPL in bits 0 and 1. Thus, a task could store the CS or SS in a general-purpose register and examine the contents of that register to find that it is not operating at the expected privilege level. The MOV opcode that loads segment registers does offer some protection because it does not allow the CS register to be loaded at all. However, if the task tries to load the SS register, several privilege checks occur that become a problem when the VM is not operating at the privilege level at which a VMOS is expecting–typically 0.

The analysis of Section 3 shows that the Intel processor is not virtualizable according to Goldberg's hardware rules.

4 Pentium-Based "VMM" Security

This section will examine several security issues for a VMM designed for the Intel Pentium architecture. We begin with a brief review of previous secure VMMs. Second, use of Intel processors for highly secure systems is discussed. Third, ways to provide virtual machine monitors on unmodified Intel platforms are examined to gain insight into the challenges faced in a virtual machine monitor effort. Next we discuss the security impact of using unmodified Intel platforms for VMMs. Finally, a better approach to creating a highly secure VMM on the Intel architecture is covered.

4.1 Are Secure VMMs Possible?

An early discussion of VMMs and security argued that the isolation provided by a combined VMM/OS provided better software security than a conventional multiprogramming operating system. It was also suggested that the redundant security mechanisms found in the VMM and the OS executing in one of its virtual machines enhanced security [21]. Penetration experiments indicated that redundant weak implementations are insufficient to secure a system [5, 11].

KVM/370 was an early secure Type I VMM [11, 33, 9]. Called a "security retrofit," two approaches to the work were examined: (1) "hardening" of the existing VM/370 control program (CP) to repair identified penetration vulnerabilities and (2) a redesign of the VM/370 CP to place all security-relevant functionality within a formally verified security kernel based upon the reference monitor concept [4]. (Note that the first approach was abandoned because flaw remediation did not provide a guarantee of the absence of yet undetected, exploitable security flaws.) The redesigned system consisted of four domains:

1. A minimized security kernel and verified trusted processes executing in supervisor state.
2. Semi-trusted processes executing in real problem state. These processes managed some global data, were audited, had access only to virtual addresses.
3. Non-kernel control programs (NKCPs) that executed the non-security relevant bulk of the VM/370 control program in real problem state. Each NKCP executed at a single security level and had access only to virtual addresses.
4. User VMs executing in real problem state under the control of a NKCP, with the same security level as the NKCP.

A security kernel is defined as hardware and software that implements the reference monitor concept [4]. A reference monitor enforces authorized access relationships between the subjects and objects within a system. It imposes three design requirements on its implementations:

1. The mechanism must be tamperproof.
2. The mechanism must always be invoked.
3. The mechanism must be small enough to be to subject to analysis and tests to ensure completeness.

The VAX Security Kernel[17] was a highly secure Type I VMM. The system's hardware, microcode, and software were designed to meet TCSEC Class A1 assurance and security requirements [22]. The project also maintained standard VMS and Ultrix-32 interfaces to run COTS operating systems and applications in virtual machines.

The VAX VMM security kernel allowed multiple virtual machines to run concurrently on a single VAX system. It could support a large number of simultaneous users and provided isolation and controlled sharing of sensitive data.

The VAX processor, much like the Intel Pentium processor, contained several sensitive, unprivileged instructions. It also had four rings. The security kernel designers modified the VAX processor microcode to make it

virtualizable. The four instructions that prevented virtualization on the VAX processor were: CHM, REI, MOVPSL, and PROBE [13]. The CHM instruction switches to a mode of equal or increased privilege. The REI instruction switches to a mode of equal or decreased privilege. The MOVPSL instruction is used to read the Processor Status Longword (similar to the machine status word in the Intel architecture). The PROBE instruction is used to determine the accessibility of a page of memory. These four instructions read or write one of the following pieces of sensitive data: the current execution mode, the previous execution mode, the modify bit of a page table entry, and the protection bit of a page table entry.

To support compatibility with existing operating systems and applications, some of the microcode changes included: defining a new VM mode bit, defining a new register called VMPSL, defining a VM-emulation exception, as well as the four instructions described above.

Ring compression, implemented entirely in software, was used to avoid certain processor modifications. The protection between compressed layers is weakened; however, this choice had little security impact since, although the VMS operating system for the VAX used all four rings, all three inner rings were in fact used for fully trusted operating system software.

The VAX I/O hardware was difficult to virtualize because its I/O mechanisms read and write various control and status registers in the I/O space of physical memory. To overcome this difficulty, the VAX security kernel I/O interface used a special, performance-optimized kernel call mechanism. To use this mechanism, a virtual machine executed a Move To Privileged Register (MTPR) instruction to a special kernel call register. The MTPR instruction trapped the security kernel software that performed the I/O. Untrusted device drivers were written for each guest OS in order to run on the VMM.

The VAX security kernel applied mandatory and discretionary access controls to virtual machines. The kernel assigned every virtual machine an access class consisting of a secrecy class (based on the Bell and LaPadula model [6]) and an integrity class (based on the Biba model [7]). The kernel supported access control lists on all objects including real devices, disk and tape volumes, and security kernel volumes. The VMM security kernel differed from a typical secure operating system because the subjects and objects are virtual machines and virtual disks, not files and processes, which are implemented by each guest OS.

It is worth noting that timing channels in VMMs [33] were addressed in the context of the VAX VMM work [14]. Despite the challenge of timing channel mitigation, VMMs provide a solution to the problem of sharing while running legacy or commercial code securely with firewalling between the VMs managed by a highly secure VMM kernel.

The VAX security effort lead to several conclusions: (1) Every ring of a processor can be emulated, but this is often not necessary. (2) Emulating a start I/O instruction is simpler and cheaper than emulating memory-mapped I/O. (3) Defining the VM as a particular processor or family of processors makes the VM more portable than if it were a reflection of the actual hardware. For example, if a VM is defined to be a Pentium processor, the VM will work on a Pentium II or Pentium III processor. (4) VM performance suffers when sensitive instructions are forced to trap to emulation software. (5) There are alternatives to modifying the microcode support for every privileged instruction to meet the needs of the VMM. (6) If a VMM is a security kernel, dependencies between the VMM and VMs must be scrutinized.

The Alpha architecture is designed to support virtualization [2]. It is designed to contain no errors that would allow protection mechanisms to be bypassed. Even "UNPREDICTABLE" results or occurrences are constrained so that security and virtualization are supported. The processor may hold or loose information as a result of "UNDEFINED" operations; however, these operations can only be triggered by privileged software. Privileged Architecture Library code (PALcode) provides a non-microcoded interface for privileged instructions. All privileged instructions must be implemented in PALcode and may be processor-model specific. The Alpha architecture supports PALcode replacement, thus allowing per-OS code to yield high performance. A VMM on the Alpha would have PALcode for all supported operating systems.

Unlike the Alpha which explicitly forbids state data from registers to be spread, most processors permit leakage of information from unpredictable results. The multiple address spaces of the Intel x86 architecture family allows such leakage [35].

The observations in this section should be considered in any attempt to design a secure Type I VMM for the Intel Pentium architecture.

4.2 Pentium Security Support

Intel 80x86 processors provide support for well understood requirements of secure systems [32]. These include call gates, segmentation, several hardware privilege levels and privileged instructions [15]. The combination of segmentation and rings are particularly supportive of secure system design and implementation [34]. The processor family was the choice for past and present trusted systems: the Boeing MLS Lan (A1)[3] [23], Gemini Trusted Network Processor

[3] TCSEC evaluation classes are given in parentheses.

(A1)[27], Verdix VSLAN (B2) [26], TIS Trusted Xenix (B2)[25], and the XTS-300 (B3) [24].

4.3 Pentium Virtualization Techniques

Since the Intel Pentium architecture is not truly virtualizable, current VMMs for the hardware base [36] must use a bit of "trickery" to realize a VMM. Each method must detect sensitive but unprivileged instructions before they are executed by a VM.

4.3.1 Pure Emulation

Pure emulation allows one system architecture to be mapped into another system architecture. By modeling a large part of the x86 instruction set in software, emulation allows x86 operating systems and applications to run on non-x86 platforms [19]. The disadvantage of emulation is significant performance degradation; no instructions are ever executed directly on the hardware. The performance degradation in Java compilers bears witness to this observation. Advances in compiler technology can help, but without specialized machine support, performance will never achieve that of a Type I VMM on a comparable hardware base. Advanced techniques, such as dynamic translation, can improve performance. Dynamic translation allows sequences of small, x86 architecture code to be translated into native-CPU code "on-the-fly." Since the native code is cached or even optimized, it can run significantly faster. This is the approach used by Transmeta [18], which provides pure emulation and is a complete software interpreter machine (CSIM) [12]. The use of register shadowing and soft memory may permit support of VMM technology. It is worth pointing out that in such systems, the morphed code must be protected from tampering or leakage of secrets. It is not clear whether such security concerns are addressed in the current generation of binary translation systems.

4.3.2 OS/API Emulation

Applications normally communicate with an operating system with a set of APIs. OS/API emulation [20] involves intercepting and emulating the behavior of the APIs using mechanisms in the underlying operating system. The out-of-kernel OS emulation used for certain Mach architectures [29] might be considered a variant of this approach. This allows applications designed for other x86 operating systems to be run. This strategy is used in Wine which provides "an implementation of the Windows 3.x and Win32 API on top of X and Unix" [37]. Wine has a program loader that allows unmodified Windows 3.1/95/NT binary files[4] to run on Intel x86-based Unix machines, such as Linux, FreeBSD, and Solaris. It allows application binaries files to run natively and achieves better performance than the pure emulation technique described above. However, OS/API emulation only works on members of the x86 OS family for which the APIs have been emulated. Furthermore, OS/API emulation is very complex. A VMM is less complicated and requires fewer updates with each new release of the OS.

4.3.3 Virtualization

A third technique is virtualization. Most hardware is only designed to be driven by one device driver. The Intel Pentium CPU is not an exception to this rule. It is designed to be configured and used by only one operating system. Features and instructions of the processor designed for applications are generally not a problem for virtualization and can be executed directly by the processor. A majority of a processor's load comes from these types of instructions. However, as discussed above, certain sensitive instructions are not privileged in the Intel architecture, making it difficult for a VMM to detect when they are executed. A strategy for virtualizing the Intel architecture would be as follows:

- Non-sensitive, unprivileged application instructions can be executed directly on the processor with no VMM intervention.
- Sensitive, privileged instructions will be detected when they trap after being executed in user mode. The trap should be delivered to the VMM that will emulate the expected behavior of the instruction in software.
- Sensitive, unprivileged instructions must be detected so that control can be transferred to the VMM.

The hardest part of the virtualization strategy is handling the seventeen problem instructions described in Section 3. Lawton describes how this is accomplished for FreeMWare[20].[5] It analyzes instructions until one of the following conditions is encountered:

1. A problem instruction.
2. A branch instruction.

[4] MS-DOS, Windows 3.1, Windowds 95, Windows 98, and Windows NT 4.0 are all trademarks of the Microsoft Corporation. All other tradmarks, including Red Hat Linux, Caldera OpenLinux, SuSE Linux, FreeBSD, and Solaris are trademarks of their respective owners.

[5] As of March 23, 2000, FreeMWare is called Plex86.

3. The address of an instruction sequence that has already been parsed.

If 1 or 2 is encountered, a breakpoint must be set at the beginning of the problem or branch instruction. If 3 is encountered, execution continues normally since this code has been analyzed already and necessary breakpoints have been installed. The complexity of this approach may render a highly secure VMM unachievable.

Code is allowed to run natively on the processor until it reaches a breakpoint. If the breakpoint occurred because of a problem instruction, its behavior is emulated by the VMM. If the breakpoint occurred because of a branch instruction, it is necessary to single step through its execution and begin analyzing instructions again at the branch target address. If the target address is not computed and has already been analyzed and marked as safe, then the branch instruction can also be marked as safe and it can run natively on the processor on subsequent accesses. Computed branch addresses require special attention. These instructions must be dynamically monitored to ensure that execution does not branch to unanalyzed code. A table might be used to keep track of the breakpoints.

Some instructions may write into memory, possibly into the address of instructions that have already been analyzed and marked as safe. The paging system is used to prevent this by write protecting any page of memory in the page tables that has already been analyzed and marked as safe. All page entries that point to the physical page with analyzed code would have to be write protected since multiple linear addresses can be mapped to the same physical page. When a write-protect page fault occurs, the VMM can unprotect the page and step through the instructions. A breakpoint can be installed before any problematic instructions. Finally, the page should be write-protected again. Instructions that cross page boundaries involve tow write-protected pages. Tables are used to track previously analyzed instructions.

Also pass-through I/O devices, timing issues, and virtualizing descriptor loading must be addressed.

Pass-Through I/O Devices: It may be useful to allow a device driver in the guest OS to drive hardware for a device that is not supported by the host OS. For example, a Linux host OS will not support a Winmodem. Pass-through devices allow a guest OS to communicate with devices using a pass-through mechanism that handles I/O reads and writes. Because control of the real hardware is turned over to the VMOS, pass-through I/O devices render security problematic.

Timing: A VMM must accurately emulate system timers. Every time slice of native code execution is bounded by an exception generated by the system timer when the execution time slice is over. The exception vectors to a routine defined in the VMM's IDT for a guest OS. A mechanism is needed that measures the time between these exceptions to emulate an accurate timer. On Intel Pentium processors, performance monitoring could be used. The RDTSC, Read Time Stamp Counter, instruction gives an accurate time stamp reading. The instruction is also executable in CPL 3, allowing efficient use in user-level VMM code.

Virtualization of Descriptor Loading: For two reasons a Pentium-based VMM must have its own set of LDT, GDT, and IDT tables. First, it allows the segment register mechanisms to work naturally. Second, it allows the VMM to have its own set of exception handlers.

Since all privilege levels (0-3) in a VM are mapped into CPL 3, the CPL is not sufficient when trying to load code that is more privileged (i.e. numerically less) than CPL 3. CPL 3 code can load descriptors as expected as long as the GDTR and LDTR registers point to the guest OS's descriptor tables. When running system code in CPL 3, exceptions are generated when loading a descriptor with that has CPL < 3. This does not occur when system code is executed at CPL 0. To solve this problem, one must trap and emulate instructions that load the segment registers when running at CPL < 3. All instructions that examine segment registers with PL < 3 must be virtualized because they may look at the RPL field.

A private GDT and LDT for the virtualization of code at CPL < 3 can also help solve this problem. Since, the instructions that reference the GDTR and LDTR are emulated, they can be loaded with values that point to the private GDT and LDT. The private descriptor tables would start out empty and generate exceptions when a segment register loads. When this happens, a private descriptor is generated that allows the next segment register load to execute natively. Every time the GDTR and LDTR are reloaded, the private descriptor tables are cleared.

4.3.4 Other Virtualization Considerations

Disco is an implementation of a Type I VMM for the Flash multi-processor [8]. It runs several different commercial operating systems on virtual machines to provide high-performance system software. Some of the key insights of the Disco implementation applicable to virtualizing the Intel Pentium architecture are described below.

Virtual CPUs: Multiple VMMs are multiplexed onto a common physical processor by using virtual processors. A data structure is kept for each virtual CPU that contains register contents, TLB contents, and other state information of the virtual CPU when it is not running on the real CPU. The VMM is responsible for managing the virtual CPUs and ensures that the effects of traps are handled properly by the executing virtual processor.

Virtual Physical Memory: To virtualize physical memory, an extra level of address translation that maintains VM physical-to-machine address mappings is used. Virtual machines are given physical addresses that start at address zero and continue to the size of the VM's memory. These physical addresses could be mapped to machine addresses used by the Intel processor using the hardware-reloaded TLB of the Intel processor. The VMM protects and manages the page table. When the VMOS tries to insert a virtual-to-physical mapping in the TLB, the VMM emulates this by translating the physical address into the corresponding VM address and inserting this into the TLB.

Virtual I/O Devices: The VMM must intercept device accesses from virtual machines and forward them to physical devices. Instead of trying to use every device's real device driver, one special device driver for each type of device is used. Each device has a monitor call that is used to pass all command arguments to the VMM in a single trap. Many devices such as disks and network interfaces require direct memory access (DMA) to physical memory. Normally these device drivers use parameters that include a DMA map. The VMM must intercept these DMA requests and translate physical addresses into machine addresses.

We note that since the VMM must control devices, a VMM for the Intel Pentium architecture must be provide device drivers for each VMOS. Loadable drivers would be particularly convenient.

Virtual Network Interface: So that VMs can communicate with each other, they use standard distributed protocols such as NFS. Disco manages a virtual subnet that allows this communication. A copy-on-write strategy for transferring data between VMs reduces the amount of copying. Virtual devices use Ethernet addresses and do not limit the maximum transfer unit of packets.

4.4 Unmodified Pentiums: VMM Security Concerns

To be a high-assurance secure computing system, security policies are correctly enforced, even under hostile attack. Examples of such systems are at least TCSEC Class B2 or an equivalent level in the Common Criteria [1]. The systems' protection mechanisms must be structured and well-defined. When dealing with highly sensitive information, labels are needed to order information into equivalence classes. Also, for environments where users are also categorized into equivalence classes based on clearances or other ordering techniques, a very effective protection mechanism is needed.

Current VMMs for the Intel architecture do not meet these requirements although some vendors claim security as a feature [31]. One claims that their product can "isolate and protect each operating environment, and the applications and data that are running in it" [36]. Another claim is that the system does "not make any assumptions concerning the software that runs within the virtual machine. Even a rogue application or operating system is confined..." Given such claims, it is worthwhile to ask how well current VMMs can enforce the VM isolation needed to support a mandatory security policy. Note that this analysis is based on assumptions regarding how virtualization is being accomplished. The following sections describe some potential problems if such systems were to be used to separate mandatory security levels.

4.4.1 Resource Sharing

A problem results from resource sharing between virtual machines. If two virtual machines have access to a floppy drive, information can flow from one VM to the other. Files could be copied from one VM to the floppy, thus giving the other VM access to the files.

4.4.2 Networking and File Sharing

A similar problem results from support of networking and file sharing. Here two virtual machines at different security levels could communicate information. Exploitable mechanisms include Microsoft Networking, Samba, Novell Netware, Network File System, and TCP/IP. For example, using TCP/IP, a VM could FTP to either a host OS or guest Linux OS and transfer files.

4.4.3 Virtual Disks

The ability to use virtual disks is also a problem. A virtual disk is a single file that is created in the host OS and used to encapsulate an entire guest disk, including an operating system and its applications. Anyone with access to this file in the host operating system could copy all information in the virtual disk to external media. The attacker could then install the virtual machine monitor on his own system and open the copied virtual disk.

Another problem is that any host OS application with read access to the file containing the virtual disk can examine the contents of virtual disk. For example, host OS file utilities such as grep can be used to search for specific strings in the virtual file system. Our tests using a Linux host OS and a Windows NT guest OS showed that a sensitive string could be located by grep in seconds on an approximately 300 MB virtual disk.

Both problems could be remedied by restricting access to the virtual file. Yet, to achieve this with any measure of assurance, a secure host OS is required.

4.4.4 Program Utilities

Tools for virtual machine interoperation may cause problems. For example, after installing VMware-Tools [36] in a guest OS, the cursor can move freely between the host OS desk-top and those of the VMs. Another feature is the ability to cut and paste between virtual machines using a feature similar to the Windows clipboard. The potential security danger if virtual machines were running at different mandatory security levels is obvious.

4.4.5 Host Operating System

For a Type II VMM, many security vulnerabilities emerge due to the lack of assurance available in the underlying host operating system. Flaws in host OS design and implementation will render the virtual machine monitor and all virtual machines vulnerable.

4.4.6 Serial and Printer Ports

Implementation of serial and printer ports presents another security problem. Before starting a virtual machine, a configuration of the guest OS must be loaded or created. A configuration option for parallel and serial ports is to have output of all parallel/serial ports go to a file in the file system of host OS. Thus on the guest OS, user attempts to print will result in output to a host OS file. Users could easily transfer information so that others could read the printer file in the host OS if its permissions were not managed carefully.

4.5 Intel-Based VMM for High Security

We conclude that current VMMs for the Intel architecture should not be used to enforce critical security policies. Furthermore, it would be unwise to try to implement a high assurance virtual machine monitor as a Type II VMM hosted on a generic commercial operating system. Layering a highly secure VMM on top of an operating system that does not meet reference monitor criteria would not provide a high level of security.

Yet the Intel Pentium processor architecture has many features that can be used to implement highly secure systems. How can these be applied?

A better approach would be to build a Type I VMM as a microkernel. The secure microkernel could be very small, making it easier for the VMM to meet the reference monitor verifiability requirement. The use of minimization, rigorous engineering, and code correspondence contribute to ensuring that the implementation is free of intentional as well as accidental flaws.

The Type I VMM would provide virtual environments on the machine. It would intercept all attempts to handle low-level hardware functions from the VMs and would control all of the devices and system features of the CPU. The microkernel could allow each VM to choose among a specific set of virtual devices, which may or may not map directly to the real devices installed on the system.

There are two advantages to using a Type I VMM to separate mandatory security levels. First, a Type I VMM can provide a high degree of isolation between VMs. Second, existing popular commercial operating systems for the processor and their applications can be run in this highly secure environment without modification. A VMM eliminates the need to port software to a special secure platform and supports the functionality of current application suites.

The biggest disadvantage to a Type I approach is that device drivers must be written for every device. This is a problem because of the wide variety of peripheral types and models available. (Note that a less secure Type II VMM avoids this problem by using existing drivers written for the host OS.) This disadvantage can be overcome when developing a secure solution by only supporting certain types and manufacturers of devices. It is not out of the ordinary for highly secure solutions to require specific types of hardware.

Before trying to implement a secure Type I VMM for the Pentium, it might be advantageous to modify the chip. Two alternative modifications could make virtualization easier. First, all seventeen unprivileged, sensitive instructions of the Intel architecture could be changed to privileged instructions. All instructions would trap naturally and the VMM could emulate the behavior of the instruction. However, this solution may cause problems in current operating systems because these seventeen instructions would now trap.

An alternative is to implement a trap on op-code instruction [12]. A new instruction is added that allows an operating system to declare instructions that should be treated as if they were privileged. This makes virtualization easier without affecting current operating systems.

Other virtualization approaches require additional code to force sensitive, unprivileged instructions to be handled by VMM software. As a result, two security concerns arise. First, the security kernel may not be considered minimal because of the extra virtualization code. Second, virtualization of the unmodified processor requires checking every instruction before it executes. Such checking is likely to doom to failure creation of a high assurance VMM.

5 Conclusions and Future Work

The feasibility of implementing a secure virtual machine monitor on the Intel Pentium has been explored.

VMM types and their hardware requirements were reviewed. Then, a detailed study of the virtualizability of all 250 Pentium instructions was conducted to determine if the processor could meet the hardware requirements of any type of VMM. The analysis showed that seventeen instructions did not meet virtualization requirements because they were sensitive and unprivileged.

After defining a strategy to "virtualize" the Pentium architecture, an analysis was conducted to determine whether a Pentium-based secure virtual machine monitor is able to securely isolate classified from unclassified virtual machines could be built. We conclude that current VMM products for the Intel architecture should not be used as a secure virtual machine monitor.

The Intel Pentium processor family already has many features that support the implementation of highly secure systems. Slight modifications to the processor would significantly facilitate development of a highly secure Type I VMM.

An effort is currently underway to examine the Intel IA64 architecture to determine how its new relate to the construction of secure systems and virtualization. The possible use of virtualization techniques for processors supporting fast binary translation is also being explored.

Acknowledgements

The authors wish to acknowledge the insight, guidance and suggestions made by Steve Lipner as this research progressed. We are grateful to Dr. Paul Karger for careful review of our manuscript, suggestions and encouragement. We wish to thank James P. Anderson for unflagging encouragement of our work and Timothy Levin for insightful discussions and review of the paper. We are grateful to the Department of the Navy for its support of the Naval Postgraduate School Center for Information Studies and Research, which made this research possible.

References

[1] ISO/IEC 15408 - Common Criteria for Information Technology Security Evaluation. Technical Report CCIB-98-026, May 1998.

[2] Alpha Architecture Handkbook. Technical Report Order Number: ECQD2KC-TE, October 1998.

[3] E. R. Altman, D. Kaeli, and Y. Sheffer. Welcome to the Opportunities of Binary Translation. *IEEE Computer*, 33(3):40–45, March 2000.

[4] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972. (Also available as Vol. I,DITCAD-758206. Vol. II, DITCAD-772806).

[5] C. Attanasio, P. Markenstein, and R. J. Phillips. Penetrating an Operating System: a Study of VM/370 Integrity. *IBM Systems Journal*, 15(1):102–116, 1976.

[6] D. E. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, MITRE Corp., Bedford, MA, 1973.

[7] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, MITRE Corp., 1977.

[8] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scaleable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[9] B. Gold, R. Linde, R. J. Peller, M. Schaefer, J. Scheid, and P. D. Ward. A security retrofit fo vm/370. In R. E. Merwin, editor, *National Computer Conference*, volume 48, pages 335–344, New York, NY, June 1979. AFIPS.

[10] B. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in Retrospect. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 13–23, Oakland, CA, April 1984. IEEE Computer Society Press.

[11] B. Gold, R. R. Linde, M. Schaefer, and J. F. Scheid. Vm/370 security retrofit program. In *Proceedings 1977 Annual Conference*, pages 411–418, Seattle, WA, October 1977. A.C.M.

[12] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. Ph.D. thesis, Harvard University, Cambridge, MA, 1972.

[13] J. Hall and P. T. Robinson. Virtualizing the VAX Architecture. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 380–389, Toronto, Canada, May 1991.

[14] W.-M. Hu. Reducing Timing Channels with Fuzzy Time. In *Proceedings 1991 IEEE Symposium on Research in Security and Privacy*, pages 8–20. IEEE Computer Society Press, 1991.

[15] Intel. *Intel Architecture Software Devloper's Manual, Volume 1: Basic Architecture*. Intel Corporation, Santa Clara, CA, 1999.

[16] Intel. *Intel Architecture Software Devloper's Manual, Volume 2: Instruction Set Reference*. Intel Corporation, Santa Clara, CA, 1999.

[17] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A Retrospective on the VAX VMM Security Kernel. *Transactions on Software Engineering*, 17(11):1147–1165, November 1991.

[18] A. Klaiber. The Technology Behind Crusoe^{TM} Processors. Transmeta Corporation, Santa Clara, CA, January 2000. also http://www.transmeta.com.

[19] K. Lawton. http://www.bochs.com, July 1999.

[20] K. Lawton. Running Multiple Operating Systems Concurrently on the IA32 PC Using Virtualization Techniques. http://www.freemware.org/research/paper.txt, June 1999.

[21] S. E. Madnick and J. J. Donavan. Application and Analysis of the Virtual Machine Approach to Information System Security. In *ACM SIGARCH-SYSOPS Workshop on Virtual Computer Systems*, pages 210–224, Boston, MA, March 1973. A.C.M.

[22] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, December 1985.

[23] National Computer Security Center. *Final Evaluation Report: Boeing Space and Defense Group, MLS LAN Secure Network Server System*, 28 August 1991.

[24] National Computer Security Center. *Final Evaluation Report of HFSI XTS-200*, CSC-EPL-92/003 C-Evaluation No. 21-92, 27 May 1992.

[25] National Computer Security Center. *Final Evaluation Report: Trusted Information Systems, Inc. Trusted XENIX Version 4.0*, January 1994.

[26] National Computer Security Center. *Final Evaluation Report: Verdix Corporation VSLAN 5.1/VSLANE 5.1*, 11 January 1994.

[27] National Computer Security Center. *Final Evaluation Report of Gemini Computers, Incorporated Gemini Trusted Network Processor, Version 1.01*, 28 June 1995.

[28] G. Popek and R. Goldberg. Formal Requirements for Virtualizable 3rd Generation Architectures. *Communications of the A.C.M.*, 17(7):412–421, 1974.

[29] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A system software kernel. In *Proceedings of the 34th Computer Society International Conference COMPCON 89*, San Francisco, CA, February 1989. IEEE Computer Society Press.

[30] J. S. Robin. Analyzing the Intel Pentium's Capability to Support a Secure Virtual Machine Monitor. Master's thesis, Naval Postgraduate School, Monterey, CA, September 1999.

[31] M. Rosenblum. Lecture at Stanford University. 17 August 1999.

[32] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[33] M. Schaefer and B. Gold. Program Confinement in KVM/370. In *Proceedings 1977 Annual Conference*, pages 404–410, Seattle, WA, October 1977. A.C.M.

[34] M. D. Schroeder and J. H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Comm. A.C.M.*, 15(3):157–170, 1972.

[35] O. Sibert, P. A. Porras, and R. Lindell. The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 211–222, Oakland, CA, May 1995. IEEE Computer Society Press.

[36] VMware Inc. *Welcome to VMware, Inc – Virtual Platform Technology*, March 1999. http://www.vmware.com/standards/index.html.

[37] Wine. http://www.winehq.com, June 2000.

Detecting and Countering System Intrusions Using Software Wrappers

Calvin Ko
Timothy Fraser
Lee Badger
Douglas Kilpatrick
NAI Labs, Network Associates, Inc.
{*calvin_ko, tfraser, lee_badger, douglas_kilpatrick*}*@nai.com*

Abstract

This paper introduces an approach that integrates intrusion detection (ID) techniques with software wrapping technology to enhance a system's ability to defend against intrusions. In particular, we employ the NAI Labs Generic Software Wrapper Toolkit to implement all or part of an intrusion detection system as ID wrappers. An ID wrapper is a software layer dynamically inserted into the kernel that can selectively intercept and analyze system calls performed by processes as well as respond to intrusive events. We have implemented several ID wrappers that employ three different major intrusion detection techniques. Also, we have combined different ID techniques by composing ID wrappers at run-time. We tested the individual and composed ID wrappers using several existing attacks and measured their impact on observed application performance. We conclude that intrusion detection algorithms can be easily encoded as wrappers that perform efficiently inside the kernel. Also, kernel-resident ID wrappers can be easily managed, allowing cooperation among multiple combined techniques to enforce a coherent global ID policy. In addition, intrusion detection algorithms can benefit from the extra data made accessible by wrappers.

1 Introduction

Intrusion detection is a retrofit approach to enhancing the security of computer systems. It utilizes various audit data to identify activities that could compromise the security of a system. Traditionally, intrusion detection systems (IDS) are user-space applications that utilize audit data generated by audit systems (e.g., Solaris Basic Security Module (BSM)) or network sniffers to detect intrusive activities. The capabilities of these user-space IDSs are restricted by the quality of the audit data and the services provided by the operating systems. For instance, audit systems do not provide all the data required by IDSs, thus limiting the attacks that can be detected by the IDSs. In addition, audit systems offer rudimentary methods for selecting data to be logged. In particular, most audit systems do not support selection of a particular program to audit. Also, as the data is generated in the kernel, every time a system action has to be logged or analyzed, the information has to be transferred from kernel space to user space, causing a context switch, and increasing the load imposed on the system by the IDS. Thus, user-space IDSs suffer from high overheads and low efficiency, as well as long delay (in CPU cycles) in detecting intrusions. Lastly, user-space IDSs are not sufficiently protected by operating systems and cannot completely protect themselves.

Our goal is to integrate ID functions into the kernel to remedy some of the problems arise in user-space intrusion detection. Specifically, we exploit the execution environment provided by Generic Software Wrappers [4] to enhance the intrusion detection and response capability of a system. An ID logic implemented as an ID wrapper can 1) selectively examine any parameters of system calls and the entire system state, 2) analyze a system call before or immediately after the call is executed, 3) analyze system calls inside the kernel, thus avoiding the overhead of transferring audit data from kernel space to user space, and 4) protect itself by denying intrusive operations.

*This research was supported by the Defense Advanced Research Projects Agency under contract F30602-96-C0333.

We have implemented several intrusion detection techniques, tested the ID wrappers using several existing attacks, and measured the performance of the ID wrappers. Our conclusion is that intrusion detection algorithms can be easily encoded as wrappers that perform efficiently inside the kernel. Also, ID wrappers can be configured and managed easily to support a coherent global intrusion detection and response policy. We envision that ID wrappers can be used individually to protect a system or as components of a large-scale intrusion detection system.

The rest of the paper is organized as follows. Section 2 presents an overview of ID wrappers, focusing on the capability of ID wrappers provided by the Generic Software Wrapper Toolkit and our extensions to the toolkit for supporting intrusion detection. In section 3, we present how we implement various ID techniques—specification-based, signature-based, and sequence-based techniques—using wrappers. In section 4, we present our experiments for testing ID wrappers with simulated attacks. We also describe a composition experiment in which two ID wrappers employing two different techniques cooperate with another abstract wrapper that combines the findings of the two ID wrappers. In addition, we present the performance results of the ID wrappers, showing that intrusion detection functions can be executed, managed, and coordinated in the kernel with a minimal observed application performance penalty. Section 5 discusses related work. In section 6, we discuss the pros and cons of the kernel-resident intrusion detection approach as well as our experience in realizing this approach using Generic Software Wrappers. Section 7 provides the conclusion and suggests future research.

2 Intrusion Detection Wrappers

This section presents the architecture of ID wrappers. It describes the capability of ID wrappers naturally provided by the Generic Software Wrapper Toolkit and our extensions to the toolkit for supporting intrusion detection.

Figure 1 gives a high-level view of an ID wrapper. An ID wrapper is a state machine that is bound dynamically to a program in execution and that gains control when system calls are invoked. Multiple ID wrappers may be bound concurrently to a single program in order to combine multiple ID techniques or to collaborate in the enforcement of a single policy. An ID wrapper is specified using the Wrapper Definition Language (WDL)[10], a superset of C language. WDL supports high-level specification of the events to be intercepted and accesses to parameters of the intercepted system call. WDL also hides specific details of different operating systems so that generic wrappers that run on multiple platforms can be written. An ID wrapper specified in WDL is compiled by the Wrapper Compiler (WrapC) into native object code of the destination platform for deployment. Currently, the wrapper toolkit supports FreeBSD, Solaris, Linux, and Windows NT[1]. ID wrapper capabilities, deriving from WDL features, fall naturally into two groupings:

Event Interception Criteria: An ID wrapper specifies events that it intercepts. Such events may be system calls or more "abstract" events defined and generated by other wrappers. An ID wrapper will listen to events that represent steps in attack specifications [5, 9], events defining (or deviating from) behavioral profiles [3, 8], events that attempt to subvert the intrusion detection system, or events that access system resources after a successful attack sequence. Events may contain parameters, and an ID wrapper may condition the interception of the events based on pre-established groupings (e.g., *open*, *close*, *read*, *write* are all "file" events), parameter value matching, global system state, and event sequence relationships (e.g., event e_1 that occur before event e_2 will be "listened for").

Actions: When an event is intercepted, an ID wrapper may take a variety of actions. In general, these actions serve to deny, transform, or augment the event, and perhaps also to generate new events that can be intercepted by other active wrappers. For intrusion detection and response purposes, an action will often be to update an intrusion detection model or fact base, to determine if any misuse rules have completed or if the current behavior exceeded the defined bounds in the normal profile, and to take countermeasures if an intrusion is imminent. Such countermeasures at least will protect the intrusion detection system from tampering, but also can include a variety of techniques that prevent damage, de-

[1]The NT prototype has a different architecture from Unix prototypes. It employs library hooking techniques to intercept the Win32 API calls performed by processes.

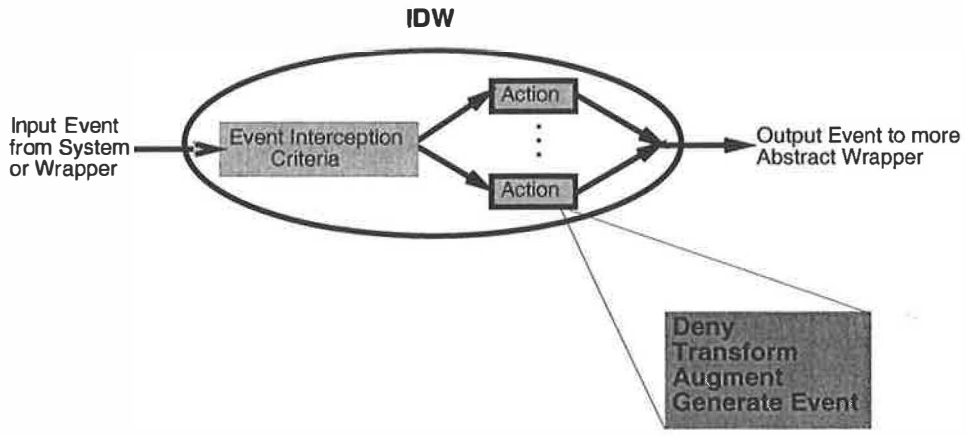


Figure 1: Intrusion Detection Wrapper Structure

ceive the intruder, or collect additional information for subsequent legal or military action. At the implementation level, ID wrapper capabilities derive from WDL facilities that support convenient access to (and modification of) event parameters, access to local environment variables and global system state, generation of new events, and access to lightweight DBMS services.

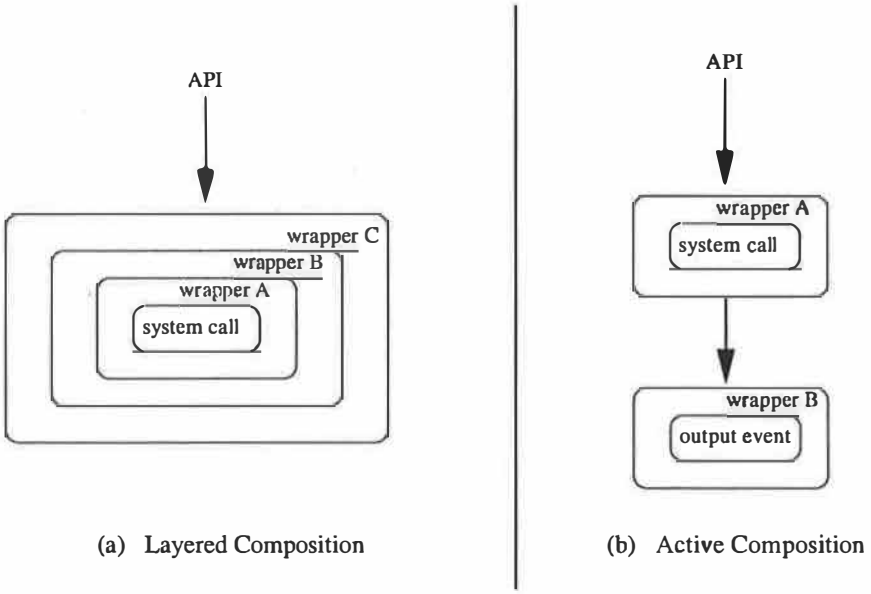
2.1 Management and Composition

ID wrappers need to be properly managed and configured to offer the best protection to a system. Depending on the overall ID policy, some ID wrappers should wrap every process while other ID wrapper should wrap only certain critical processes. The Wrapper Support Subsystem (WSS) provides support for configuration and management of ID wrappers. To use an ID wrapper, an administrator first registers the wrapper with the WSS through a loading process, which dynamically inserts the run-time image of the wrapper into the kernel. Selection of processes for wrapping is controlled by activation (or deactivation) criteria which specify when a loaded wrapper should begin (or cease) to wrap a process. The activation criteria language allows specifications based on the invoker, the program name, and attributes of the executable. The WSS tracks running processes and evaluates the activate criteria to activate wrappers to wrap processes that satisfy the criteria. Therefore, ID wrappers can be configured and administered easily in our framework to enforce a coherent ID policy.

The whole problem of intrusion detection is beyond the capability of any one intrusion detection system or ID technique [6]. Therefore, cooperation of different ID techniques is required to enhance the protection of a system. To combine multiple ID techniques, it is often convenient to implement each ID technique in a separate, independent ID wrapper and to run processes under the simultaneous control of multiple ID wrappers. Additionally, it is highly desirable to have ID wrappers that are aware of one another to support hierarchies of increasingly abstract wrappers. For example, one ID wrapper can listen to system calls to generate abstract system independent audit events to be consumed by a more abstract ID wrapper that analyzes the abstract audit events. Figure 2 shows the two fundamental forms of composition:

Layered Composition: Multiple ID wrappers intercept an event (e.g., a system call) and perform some actions. In this case, the actions of the wrappers will be executed in the order in which the wrappers were installed on the system. Figure 2a illustrates the ordering for layered composition. In layered composition, the wrappers involved in the composition might not be aware of the composition occurring. This type of layering could be compared to an onion, in which the user's request must travel down through the "layers" of wrappers to get to the system call; the return value must travel back out through the "layers" to reach the API again.

Active Composition: ID Wrappers generate events intercepted by other ID wrappers (out-



(a) Layered Composition

(b) Active Composition

Figure 2: Wrapper Composition

put events), shown in figure 2b. Output events represent active composition, in which the wrappers generating the events are aware of the possible communication/coordination with other wrappers. In this instance, a ID wrapper generates an output event to be intercepted by another, usually more abstract, ID wrapper. The more abstract wrapper will return to the calling wrapper; control passes through the calling wrapper to the system call.

The two forms of composition are not mutually exclusive: a system event could be intercepted by layers of ID wrappers, some of which could generate output events to be intercepted by other ID wrappers. The composition facility is flexible enough to allows ID wrappers to cooperate in the manners (e.g., complement or reinforce each other's findings) described by in Common Intrusion Detection Framework [6].

2.2 Obtaining System State Information

In addition to the parameters of the intercepted system calls, ID wrappers may need to access system state to acquire additional data for its analysis. For example, the owner, group, and permission mode of a file being accessed may be required to determine whether this file access deviates from a specified valid behavior profile.

An ID support module has been added to the Generic Software Wrappers toolkit to provide a set of library functions for ID wrappers. Table 1 enumerates the library functions that have been implemented. Additional functions can be implemented and added to the system easily.

2.3 Dispatching Audit Data to User-Space IDSs

In a large-scale IDS, an ID wrapper may be used as a data-collection component that collects security-relevant data for intrusion analysis engines running in user space. Such scenario requires a very efficient mechanism for transferring a large amount of data from wrappers running in kernel space to user processes in a secure fashion. In addition, such mechanism should allow multiple intrusion detection systems to listen to the audit event data generated by possibly different ID wrappers.

An audit event handler providing support for dispatching audit data to user processes is incorporated into the basic wrapper toolkit. An intrusion detection engine cooperating with an ID wrapper

Name	Function
wr_stat	obtain status of the file specified by a path
wr_fstat	obtain status of the file specified by a file descriptor
wr_audit	delivery audit data to the audit event handler
wr_audit_printf	same as wr_audit, but with the printf interface
wr_get_addr	obtain the socket address of a socket specified by a file descriptor
wr_getpeername	get the name of the peer of a connection
wr_getsockname	get the name of the local entity of a connection

Table 1: Wrapper Library functions to support Intrusion Detection

can register with the audit event handler for the event queue to which it wants to listen. When the cooperating ID wrapper collects relevant audit data and sends it to the audit event queue, the audit event handler dispatches the data to the registered intrusion detection engine.

In this approach, the IDS thread calls a registered system call to register for some number of audit queues. The system call creates a pipe and returns the read end of the pipe. The IDS thread performs a select system call on the read end of the pipe, effectively blocking the process. The event handler writes the entire event structure for each audit event to the write end of the pipe. This method can promptly transfer events from the event handler to the waiting thread in a thread-safe manner and with little overhead.

3 Implementation

This section presents our experience in the implementation of various intrusion detection techniques using ID wrappers.

3.1 Specification-based Techniques

Specification-based intrusion detection systems employ specifications that describe the valid behavior of programs to detect intrusions. In particular, programs in execution are monitored for violations from the corresponding valid behavior specifications. One useful type of specifications is the set of valid operations of a program [7]. We have encoded the specifications for several programs (e.g., *imapd, fingerd, lpr, lprm, ftpd, httpd*, etc.) describing their valid operations as ID wrappers using WDL. Each specification-based ID wrapper is configured, using the activation criteria, to wrap the execution of the program with which the specification is concerned. We found that the WDL itself is very suitable for expressing the set of valid operations of a program.

Figure 3 shows part of a specification of the *imapd* program, encoded in WDL, that is concerned with the valid parameter values of *open-read*, *chmod*, *fchmod*, and *execve* operations.

The first clause specifies that after a successful open system call with the read-only flag on (lines 1-2), the action block (lines 3-7) will be executed. The action block obtains the inode information of the opened file and checks whether 1) it is world-readable, 2) owned by the invoker, or 3) created by the program execution (checked by the local function created()). It raises a violation if all conditions are false. In short, the first clause detects any bad open-read operation: on a file that is neither publicly readable, owned by the invoker, nor created by the program execution itself. Similarly the second and third clauses raise a violation if the program performs a *chmod/fchmod* operation on a file not created by the program execution. The last clause specifies that the wrapper intercepts the execve system call before it executes, issues a violation, and prohibits the call by returning an error code to the caller immediately (`return WR_D_BADPERM`).

The partial specification illustrates that criteria for event interception can be specified very easily in WDL. In addition, accesses to system call parameters can be accomplished easily through special $ variables. For example, the $path variable on line 10 denotes the path name of the file in the *chmod* system call. A reference or assignment to a variable effectively reads/modifies the corresponding argument of the intercepted system call. WDL

```
1.  bsd::op{open}
2.  ($errno == 0 && $flags | O_RDONLY != 0) post {
3.      struct wr_stat s;
4.      wr_fstat($fdret, &s);
5.      if (!WorldReadable(s) && Owner(s) != _uid && !created(s.nodeid))
6.          violation();
7.  };
8.  bsd::op{chmod} pre {
9.     struct wr_stat s;
10.    wr_stat($path, &s);
11.    if (!created(s.nodeid))
12.        violation();
13. };
14. bsd::op{fchmod} pre {
15.    struct wr_stat s;
16.    wr_fstat($fd, &s);
17.    if (!created(s.nodeid))
18.        violation();
19. };
20. bsd::op{execve} pre {
21.    violation();
22.    return WR_D_BADPERM;
23. };
```

Figure 3: Partial imapd program behavior specification in WDL

also handles the copying of argument data between user space and kernel space automatically, allowing wrapper developers to focus on the key aspects of a wrapper instead of low-level programming details.

With ID wrappers, we can monitor programs for improper modifications to objects that otherwise cannot be accomplished using traditional audit trails. In particular, ID wrappers can examine data read/written to specific files without a huge overhead. Using this capability, we wrote an ID wrapper that examines the *passwd* program to ensure that when a user (say Joe) invokes the *passwd* program, the program modifies only the part of the password file associated with the password of Joe. If there is a vulnerability in the *passwd* program that allows the attack to control the program to arbitrarily modify the password file (e.g., changing the user ID of a user), this ID wrapper is able to detect such an attack.

3.2 Signature-based Techniques

Signature-based ID systems detect intrusions by observing events and identifying patterns which match the signatures of known attacks. An attack signature defines the essential events required to perform the attack, and the order in which they must be performed. Different ID systems represent signatures in different ways. The State Transition Analysis Tool (STAT) [5], for example, represents signatures with state transition diagrams. During runtime, these diagrams direct the operation of finite state machines that represent possible intrusions in progress. The STAT system advances these state machines from state to state as it observes events that match parts of attack signatures. If the STAT system observes a sequence of events that ultimately moves one of these finite state machines to its final state, the STAT system declares that it has detected an intrusion.

We have implemented the `Mailstat` wrapper, an example of STAT-like ID which attempts to detect a well-known attack on a commonly-used UNIX mail daemon. The signature of this mail daemon attack is effectively hard-coded in the structure of the Mailstat wrapper. When deployed, the `Mailstat` wrapper wraps all processes on the system, and intercepts and examines every system call that might correspond to an event in the mail daemon attack signature. It uses a database table to store the state of the finite state machines representing possible attacks in progress. Whenever `Mailstat` observes a system call that matches the first event in the

mail daemon attack signature, it creates a new finite state machine by adding a new line to the table. As it intercepts system calls and observes events, it advances the state of the appropriate finite state machines according to the mail daemon attack signature's state transition diagram. When any finite state machine in the table reaches its final state, the `Mailstat` wrapper indicates an intrusion and reports the identities of the processes which caused the events leading to its detection.

3.3 Sequence-based Techniques

The sequence-based intrusion detection approach by Forrest [3] calculates an anomaly value for a program execution based on the number of sequences the program generates that are missed in a precomputed database of sequences. The technique has been found to be effective under offline evaluation using audit data collected from different environments. It requires properly-constructed norma sensitive to program versions and configuration, and can in some cases require significant processing resources to perform anomaly calculation in real time. We have structured `Seq_id`, our sequence-based ID wrapper, to address these issues.

`Seq_id` runs in two modes: record mode and detect mode. In record mode, `Seq_id` automatically generates a normative sequence database for each program executed. Using `Seq_id`, we have generated a per-program database for every program executed on our test machines. To increase efficiency and simplicity, we have slightly modified the algorithm described in [2] to merge some sequences, which would remain unique in the original technique. Initial comparison tests between the two algorithms indicate that the detection accuracy is similar. In detect mode, `Seq_id` decides if each observed system call completes a sequence stored in the program's database of normal behavior. `Seq_id` measures the magnitude of each deviation, and reports those of sufficient magnitude.

4 Experiments and Performance Measurement

To evaluate the intrusion detection wrappers with respect to their ability to detect attacks, we tested the ID wrappers with several existing attacks. These attacks exploit vulnerabilities in security-critical programs that possess privileges to obtain a shell running as root. We describe the programs and the attacks below.

imapd Some versions of the Internet Mail Access Protocol (IMAP) server contain a number of buffer-overflow bugs that allow a remote user to obtain a shell running as root (CERT Advisory CA-97.09). We obtained an exploitation script to one of the bugs from RootShell (www.rootshell.com). The exploit script carefully crafts the input to *imapd* that exceeds the size of a special stack buffer and presents the name to the IMAP server to overwrite the saved instruction pointer and execute the planted machine code. The code then executes a shell running with root. We wrapped *imapd* using a specification-based ID wrapper `Imapd_id` specific to *imapd* and a sequence-based ID wrapper separately. Both wrappers were able to detect the exploit script's attack.

lpr Due to insufficient bounds checking on arguments which are supplied by users, it is possible to overwrite the internal stack space of some versions of the *lpr* program while it is executing. This can allow an intruder to cause *lpr* to execute arbitrary commands by supplying a carefully designed argument to *lpr* (AUSCERT Advisory AA-96.12). These commands will be run with the privileges of the *lpr* program. When lpr is setuid root it may allow intruders to run arbitrary commands with root privileges. We simulated the attack using a script from RootShell. We wrapped lpr using a specification-based wrapper tailored for lpr and the wrapper was able to detect the attack.

lprm The program *lprm* is part of the printing subsystem. The program is used to remove a job in the printer queue. There is a buffer-overflow vulnerability in some versions of this program that allows a local user to execute arbitrary commands with root privileges. We obtained a script from Security Bugware (http://161.53.42.3/crv/security/bugs/list.html) and tested a specification-based wrapper written for *lprm* with the script. The specification-based wrapper detected the attack when *lprm* was tricked to execute the Bourne shell.

binmail The *binmail* program is the back-end mailer that delivers mail messages to

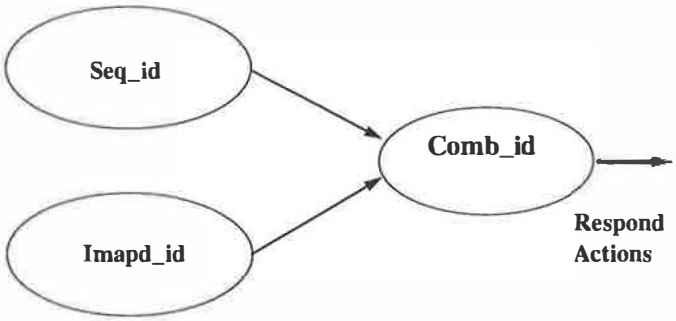


Figure 4: Composing Two ID techniques

users' mailboxes. It does so by appending the messages to the mailbox files directly. In some old versions, *binmail* changes the ownership of a user's mailbox (usually `/var/spool/mail/<username>`) back to the user after it appends a message if the mailbox file is not owned by the user initially. In particular, the *binmail* program (`/bin/mail`) in 4.2 BSD Unix fails to reset the setuid bit of the mailbox file after it appends a message and changes the owner of the file [5]. An attacker, who creates a mailbox file with the setuid bit on for the superuser, can trick *binmail* into making the file to be setuid root by invoking *binmail* to send a mail message to root. We deployed the `Mailstat` wrapper and tested the wrapper with an exploitation script we created. The wrapper detected the intrusion immediately.

4.1 Combining Multiple Techniques using Composition

Our wrapper frameworks allow multiple ID wrappers to cooperate to enhance their performance. The Common Intrusion Detection Framework [6] discusses several ways ID components cooperate with each other. We performed an experiment in which two ID wrappers cooperates to reinforce each others findings. Figure 4 depicts the configuration. A sequence-based wrapper and a specification-based wrapper are used to wrap the *imapd* programs. Every system call performed by *imapd* will be intercepted by both wrappers (The order will be determined by the loading sequence). Each wrapper will analyze the operations of *imapd* individually and generate an abstract warning event to the abstract wrapper (Com_id) when they find an attack. The abstract wrapper judges the output from both *Seq_id* and *Imapd_id* and accepts it when both ID wrappers think the program is under an attack. In this case it will kill the process.

We tested the composite ID wrappers using the *imapd* attack described in the last subsection. An interesting observation is that the two wrappers detect the attack at different system call. The `Imapd_id` detected the attack when the program executes a Bourne shell (at the *execve* system call). The `Seq_id` detected the attack several system calls after the *execve* system call. The abstract ID wrapper `Com_id` killed the process after it receives warning from both wrappers. Potentially, such configuration could reduce the false positive rate as the whole IDS will detect a false attack when both techniques produce a false positive. However, it could also cause some attacks to escape the detection if only one technique detects the attack. Thus, further research is needed to determine how to best combine different techniques.

4.2 Performance

We have studied the performance of the intrusion detection wrappers. We measured the overhead caused by the intrusion detection wrappers on the running time of programs using a Kernel Build test, in which the time taken to compile a Generic version of the FreeBSD kernel was measured. Also, we measured the overhead caused by ID wrappers from a user's perspective, in terms of latency and throughput, for a Web server and a FTP server.

	Average Kernel Build Time			Average HTTP Latency			Average HTTP Throughput		
	time (s)	σ (s)	penalty	time (s)	σ (s)	penalty	t-put (Mbits/s)	σ (Mbits/s)	penalty
no WSS	583.43	0.53	0%	0.657	0.025	0%	7.455	0.21	0%
WSS only	604.38	0.46	3.47%	0.652	0.0023	-0.081%	7.456	0.08	0.01%
Seq_id	624.62	1.23	6.59%	0.687	0.017	4.52%	7.038	0.164	5.61%
Http_id	-	-	-	0.705	0.0247	7.38%	6.928	0.157	7.08%
Http_id & Seq_id	-	-	-	0.744	0.018	13.26%	6.607	0.127	11.39%

Table 2: FreeBSD Prototype Performance for Kernel Build and Web Server Benchmarks

	Average FTP Latency			Average FTP Throughput		
	time (s)	σ (s)	penalty	t-put (Mbits/s)	σ (Mbits/s)	penalty
no WSS	28.2418	0.9019	0%	8.776	0.093	0%
WSS only	28.3332	1.0773	0.32%	8.768	0.069	0.09%
Seq_id	28.30125	1.0835	0.21%	8.743	0.076	0.38%
Ftpd_id	28.3592	0.7954	0.42%	8.756	0.085	0.23%
Ftpd_id & Seq_id	27.9224	1.2007	-1.13%	8.573	0.012	2.31%

Table 3: FreeBSD Prototype Performance for FTP Server Benchmarks

Table 2 summarizes the results of the performance tests for Kernel Build and for a Web server. The first column shows the average time taken to compile the FreeBSD kernel 1) under normal conditions, 2) with the WSS loaded into the kernel, and 3) with the sequence-based intrusion detection wrapper `Seq_id` wrapping the compilation process. The second and third columns of the table contain results for a custom-made Web server benchmark. The Average HTTP Latency column describes the delay a Web client experiences between the moment it makes a request and the moment it receives the Web server reply. The Average HTTP Throughput describes the rate at which the Web server returns data to the Web clients. We measured the latency and throughput of the Web server when the Web server is wrapped by Seq_id, Http_id, and both Seq_id and Http_id. The results were produced by a custom-made Web server benchmark executed with an Apache 1.3.0 Web server and the WebStone 2.0.1 benchmarking software. The Apache Web server ran on a 166MHz Intel Pentium-based microcomputer with 32MB RAM running a Generic FreeBSD 2.2.2 kernel. Two Pentium 400MHz machines were used to run 32 WebStone 2.0.1 Web clients through a series of 10 15-minute trials using the standard WebStone 2.0.1 file set for each row in table 2.

Table 3 shows the results of the performance tests using a custom-made FTP server benchmark. The Average FTP Latency column describes the delay a FTP client experiences between the moment it makes a anonymous request and the moment it receives all the data from the server. The Average FTP Throughput describes the rate at which the FTP server returns data to the FTP clients. The table denotes the latency and throughput of the FTP server under controls to Seq_id, Ftp_id, and both Seq_id and Ftp_id. Ftp_id is a specification-based wrapper that restricts the operations that can be performed by the FTP server. The FTP server (ftpd) in the FreeBSD 2.2.2 distribution was used in the tests. The Average FTP Latency and Average FTP Throughput results were obtained in a smiliar manner to the HTTP results using a modified WebStone software that performs anonymous FTP fetches instead of HTTP fetches.

The performance results show that WSS alone imposes 3-4% penalty on the compilation time of the FreeBSD kernel. `Seq_id` adds another 3-4% to the compilation time of the FreeBSD kernel. Impact caused by WSS on the latency and throughput of a Web/FTP server is minimal, possibly because WSS only intercept the *fork*, *execve*, and *exit* system calls, which are used infrequently in a Web/FTP server.

The sequence-based wrapper and the specification-based wrappers impose approximately 5-7 % overhead on the Web/FTP server, and their impacts add up when they are used together. While we have designed the wrapper toolkit and ID wrappers with consideration for performance, we have not optimized the prototype; therefore, performance can possibly be improved.

5 Related Work

Balasubramaniyan et. al. [1] have proposed the use of autonomous agents for intrusion detection. They have developed an architecture for the autonomous ID agents. Our idea is similar to their agent idea in that ID wrappers can be viewed as kernel-resident ID agents. Their conjecture is that the performance of the agents can be improved if they are implemented inside the kernel. Our results support their conjecture; in particular, kernel-resident agents can be very efficient and impose very little performance penalty on a system.

Sekar et. al. [11] have devised an efficient method of implementing a form of specification-based intrusion detection in the kernel. Some of the implementation strategies employed by Sekar's method are similar to those we have employed in ID wrappers. For example, both efforts associate individual kernel-resident state machines ("wrappers," in our terminology) with each application process under observation, using interposition techniques at the operating system's system call interface to enable these kernel-resident state machines to observe application process behavior at a fine-grained level of detail. Sekar's effort concentrates on the efficient implementation of a single form of specification-based intrusion detection, and has achieved a result which allows the intrusion detection system to handle multiple patterns with the same low overhead as a single pattern. Our effort, in contrast, has sought to produce a general framework for the implementation of multiple intrusion detection algorithms, as well as a convenient means for managing their simultaneous deployment and composition. Both our effort and Sekar's have observed favorably low overheads in terms of observed application performance degradation due to the use of kernel-resident intrusion detection. Sekar's technique resulted in overheads of no more than 1.5% in *ftpd*, *telnetd*, and *httpd* benchmarks documented in [2].

6 Discussion

The idea of moving intrusion detection functions into the kernel is not new and has been hinted at in the literature. Balasubramaniyan et. al. discussed the advantages and disadvantages of integrating intrusion detection agents inside the kernel [1].

In-kernel intrusion detection has several advantages. First, overhead due to extra context switching is avoided – a system call is analyzed by the ID logic at the same kernel context at which the system call executes. In addition, information is registered and processed at or near the place where it is produced, reducing the time and resources for transferring the information to the analysis engine. Also, this proximity allows prompt detection and reduces the possibility of the information being modified by an attacker before it gets to the ID analysis engine. Lastly, it is harder for an intruder to tamper with the ID system as the attacker would have to modify the kernel (e.g., by defeating the kernel's memory-protection mechanism).

However, the kernel-resident implementation strategy also has its disadvantages. First, kernel-resident ID systems are not portable across platforms. Second, a misbehaving ID system can do much more damage if it is running in the kernel rather than in user space because it has full access to the system. Third, entities in the kernel can have a large impact in the host behavior by slowing down fundamental operations (e.g., kernel data structures, accesses to memory, disk). Fourth, entities inside the kernel are very difficult to manage and configure. Finally, kernel programming is at a low level of abstraction, where the resources available provide very limited functionality when compared to the higher-level abstractions available in user space.

Our work essentially illustrates that in-kernel intrusion detection is feasible and practical provided that the kernel-resident ID system is designed and coded carefully. With minimal adjustment, many intrusion detection techniques can be implemented to run inside the kernel efficiently without impacting the host behavior. By using the Generic Software Wrapper Toolkit as the basis for implementing kernel-resident intrusion detectors, our approach inherits the advantages of in-kernel intrusion detection while avoiding the problems of portability, manageability, and the low-level nature of kernel programming. We strongly believe that further investigation of in-

kernel intrusion detection is worthwhile and necessary.

7 Conclusion and Future Work

We have described our effort to enhances IDS capability by exploiting the execution environment offered by software wrappers. In order to take advantage of the potential for increased functionality and performance in kernel-resident intrusion detection systems, we have begun the development of a Generic Software Wrapper-based ID support framework, and have explored this framework's ability to ease the implementation, management and simultaneous composed deployment of three major intrusion-detection algorithms. We have described our ID-support extensions to the basic Generic Software Wrapper Toolkit, and how these extensions eased the implementation of our prototype ID wrappers. Based on our experience and the results of our performance benchmarks, we predict that many ID techniques can be efficiently implemented as kernel-resident wrappers. In all of our benchmarks, the overall observed application performance penalty associated with the use of our ID wrappers never exceeded 7.4%.

In addition to increased efficiency, ID wrappers derive several other benefits from their kernel-resident Generic Software Wrapper-based implementation. First, the interposition capability of the wrappers system provides ID wrappers with a greater range of fine-grained event data than is available to user-space techniques which must rely upon log-based audit data. All system calls and their parameters are visible to ID wrappers. Second, this interposition capability and the generality of the C-based wrapper implementation language allows wrappers to respond to intrusive events as they occur, with a broad range of response functionality. Finally, using the wrapper framework, kernel-resident ID components can be configured and managed easily to enforce a global ID policy and possibly to interoperate with large scale IDS running in user space.

Our most promising direction for future research concerns the composition of multiple intrusion detection wrappers at run-time. The ability to simultaneously apply multiple complimentary intrusion detection techniques to the same event stream appears to present a potential means of providing more accurate detection. Another promising direction involves utilization of wrapper's ability to examine data read/written to specific files or connection endpoints (e.g., sockets) to detect attacks that cannot be spotted by just looking at parameters of system calls. Other directions include cooperation with large-scale intrusion detection systems, the development of distributed ID wrappers, and efforts to improve the trust-worthiness and safety of the kernel-resident ID module.

References

[1] J. Balasubramaniyan et al. An Architecture for Intrusion Detection using Autonomous Agents. Technical Report TR-98-05, Department of Computer Science, Purdue University, June 1998.

[2] T. Bowen, D. Chee, M. Segal, R. Sekar, T. Shanbhag, and P. Uppuluri. Building survivable systems: An integrated approach based on intrusion detection and damage containment. *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX) 2000*, 2:84–99, 2000.

[3] S. Forrest et al. A sense of self for unix processes. In *Proceedings of the 1996 Symposium on Security and Privacy*, pages 120–128, Oakland, CA, May 6-8 1996.

[4] T. Fraser, L. Badger, and Feldman M. Hardening COTS software with generic software wappers. In *Proceedings of the 1999 Symposium on Security and Privacy*, 1999.

[5] K. Ilgun, R. Kermmerer, and P. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[6] C. Kahn et al. A common intrusion detection framework. *Journal of Computer Security*, 1998.

[7] C. Ko, G. Fink, and K. Levitt. Automated Detection of Vulnerabilities in Privileged Programs Using Execution Monitoring. In *Proceedings of the 10th Computer Security Application Conference*, Orlando, FL, December 5-9, 1994.

[8] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-Based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, California, 1997.

[9] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Department of Computer Science, Purdue University, August 1995.

[10] Karen Oostendorp, Christopher Vance, Kelly Djahandari, Benjamin Uecker, and Lee Badger. Preliminary Wrapper Definition Language Specification. Technical Report #0684, Trusted Information Systems, Inc., 1997.

[11] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. *Proceedings of the 8th USENIX Security Symposium*, pages 63–78, 1999.

Detecting Backdoors

Yin Zhang
Department of Computer Science
Cornell University
Ithaca, NY 14853
yzhang@cs.cornell.edu

Vern Paxson*
AT&T Center for Internet Research at ICSI
International Computer Science Institute
Berkeley, CA 94704
vern@aciri.org

Abstract

Backdoors are often installed by attackers who have compromised a system to ease their subsequent return to the system. We consider the problem of identifying a large class of backdoors, namely those providing interactive access on non-standard ports, by passively monitoring a site's Internet access link. We develop a general algorithm for detecting interactive traffic based on packet size and timing characteristics, and a set of protocol-specific algorithms that look for signatures distinctive to particular protocols. We evaluate the algorithms on large Internet access traces and find that they perform quite well. In addition, some of the algorithms are amenable to prefiltering using a stateless packet filter, which yields a major performance increase at little or no loss of accuracy. However, the success of the algorithms is tempered by the discovery that large sites have many users who routinely access what are in fact benign backdoors, such as servers running on non-standard ports not to hide, but for mundane administrative reasons. Hence, backdoor detection also requires a significant policy component for separating allowable backdoor access from surreptitious access.

1 Introduction

A *backdoor* is a mechanism surreptitiously introduced into a computer system to facilitate unauthorized access to the system. While backdoors can be installed for accessing a variety of services, of particular interest for network security are ones that provide interactive access. These are often installed by attackers who have compromised a system to ease their subsequent return to the system.

From a network monitoring perspective, such backdoors frequently run over protocols such as Telnet [PR83a], Rlogin [Ka91], or SSH [YKSRL99]. An example of a non-interactive backdoor would be an unauthorized SMTP server [Po82], say to facilitate relaying email spam; and one somewhat in between would be an FTP [PR85] backdoor used to provide access to illicit content such as pirated software, or a Napster server [NA99] run in violation of a site's policy.

Backdoors are, by design, difficult to detect. A common scheme for masking their presence is to run a server for a standard service such as Telnet, but on an undistinguished port rather than the well-known port associated with the service, or perhaps on a well-known port associated with a *different* service. In this paper we examine the problem of detecting backdoors, particularly interactive ones, by inspecting network traffic using an intrusion detection system (IDS), where we presume that there is a large volume of legitimate traffic which must be distinguished from the illegitimate traffic. To our knowledge, this problem has not been previously addressed in the literature.

Our general approach is to develop a set of algorithms for detecting different types of interactive traffic. These algorithms can then be applied to a traffic stream and whenever they detect interactive traffic using a non-standard service port, we have found some form of backdoor.

The rest of the paper is organized as follows. In § 2, we discuss the design considerations and examine the tradeoffs of different approaches. In § 3, we develop a general algorithm for detecting interactive traffic based on its timing characteristics, and in § 4 we present a number of protocol-specific algorithms. In § 5, we evaluate the algorithms using traces of Internet traffic. We summarize in § 6.

*Also with the Lawrence Berkeley National Laboratory.

2 Design Space

A basic principle for backdoor detection is to find distinctive features indicative of the activity of interest, be it general interactive access, or use of a specific protocol such as SSH. The more powerful a feature is for distinguishing between genuine instances of the activity and false alarms, the better.

Candidates for such features include the specific contents of the data stream, the size and transmission rate of the packets in the stream, and their timing structure. This last is potentially very powerful for detecting interactive traffic: studies of Internet traffic have found that the interarrivals of user keystrokes have a striking distribution [DJCME92, PF95], namely a Pareto with infinite variance. There is also the possibility that a combination of features will prove to have greater distinctive power than any one feature by itself.

We now turn to a discussion of various tradeoffs that arise when considering how to develop detection algorithms.

2.1 Open vs. evasive attackers

In general, network intrusion detection becomes much more difficult when the attacker actively attempts to evade detection by the monitor [PN98, Pa98]. Much of the difficulty comes from the ability of attackers to exploit ambiguities in a traffic stream. From a monitoring perspective, heuristics might work well for "open" (non-evasive) attackers, but completely fail in the face of an actively evasive attacker.

While ideally any detection algorithms we develop would of course be resistant to evasive attackers, ensuring such robustness can sometimes be exceedingly difficult, and we proceed here on the assumption that there is utility in "raising the bar" even when a detection algorithm can be defeated by a sufficiently aggressive attacker. We further note that if an attacker fully controls *both* the remote and the local host, and in particular if they are patient and/or able to deploy arbitrary software, then all sorts of devious covert channels become possible[1] [Gl93], and backdoor detection becomes essentially hopeless. We do not attempt to address the problem of detecting covert channels.

[1] See [Ra00] for a discussion of experiences with running NFS over email by tunneling IP packets over messages delivered by SMTP.

Thus, we propose the algorithms in this paper not as solutions, but merely as waystations in the ongoing "arms race" between attackers and intrusion detection. One form of arms race we anticipate is particularly likely is between the developers of Napster [NA99] (and Gnutella [GN00]) and our corresponding detection algorithm. Napster has a history of sites attempting to control its use, and of users attempting to circumvent these restrictions [We00], and our algorithm gives sites a new tool for detecting surreptitious use of Napster.

2.2 Passive vs. active monitoring

One tradeoff is whether we only allow the monitor to perform passive monitoring, or if it can actively inject traffic into the network. Passive monitoring has the advantage that it cannot disturb the normal operation of the network. On the other hand, an active monitor could augment its backdoor detection by trying to connect to suspected backdoors in order to probe the server listening on the port to determine its service. However, doing so could in principle tip off the attacker as to the presence of the monitor and the discovery of the backdoor.

In this paper we confine ourselves to monitors that only use passive monitoring.

2.3 Content vs. timing

A natural approach for detecting connections to command shell servers is to monitor the keystrokes looking for common shell commands. Such a content-based approach has several drawbacks, however:

- Scanning each byte in each incoming packet is very expensive, especially if we must first reassemble TCP streams to defeat the sort of evasions characterized in [Pa98]. The intruder can then overload the monitor by generating a large amount of legitimate traffic.
- Many command shells allow the user to define aliases and editing characters, which can easily defeat this approach unless the monitor performs alias and editing expansion of the commands (such as also required for "bottleneck" analysis [LWWWG98]). Note that this problem can arise either inadvertently, because the attacker as a matter of course uses aliases or redefines the editing

sequences, or deliberately, when the attacker is attempting to evade detection. The former case may be amenable to heuristic analysis; the latter likely is not.

- The intruder can easily evade the monitor by encrypting their content either through some application-level encryption method, or directly using encrypted protocols such as SSH.

In contrast, timing-based algorithms can be completely unperturbed by the use of encryption. However, timing information can become distorted due to clock skew, propagation delays, loss, and packetization variations. Making timing-based algorithm robust against such noise is challenging.

2.4 Filtering

An important factor for the success of real-time backdoor detection is filtering. The more traffic that can be discarded on a per-packet basis due to patterns in the TCP/IP headers, the better, as this can greatly reduce the processing load on the monitor. As we will see in subsequent sections, filtering can sometimes be highly effective in winnowing down a large traffic stream to just a few packets of interest.

However, there is clearly a tradeoff between reduced system load and lost information. First, if a monitor detects suspicious activity in a filtered stream, often the filtering has removed sufficient accompanying context that it becomes quite difficult to determine if the activity is indeed an attack. In addition, the existence of filtering criteria makes it easier for the attackers to evade detection by manipulating their traffic so that it no longer matches the filtering criteria. For example, an evasion against filtering based on packet size (see below) is to use a Telnet client modified to send a large number of do-nothing Telnet options along with each keystroke or line of input.

In addition, reliance on filtering can significantly magnify the problem of "chaff," i.e., attackers generating bogus traffic that matches the filtering criteria in order to overwhelm the monitor's analysis load, and/or to generate a huge number of false positives, in order to mask a true attack.

Three possible filtering criteria for backdoor detection are:

- *Packet size.* Keystroke packets are quite small. Even when entire lines of input are transferred using "line mode" [Bo90], packet payloads will tend to be much smaller than used for bulk-transfer protocols. Therefore, by filtering packets to only capture small packets, the monitor can significantly reduce its packet capture load.

- *Directionality.* In general, an interactive connection such as Telnet is initiated by the client rather than the server, unless the attacker sets up some sort of *callback* mechanism. This makes it possible to filter connections based on their directionality (inbound vs. outbound). If we are monitoring an Internet access link and are only interested in detecting backdoors at the local site, we can limit our monitoring to just inbound connections, which can significantly reduce the packet capture load (for example, by filtering out outbound Web surfing connections).

  Note that there is also a "cold start" problem when the monitor starts running and needs to analyze an existing traffic stream. In this case, it generally cannot determine whether the traffic was initiated inbound or outbound, and accordingly cannot filter it out.

- *Packet contents.* When we are interested in identifying specific interactive protocols, it is sometimes possible to filter incoming packets based on patterns specific to the protocol. An example is SSH, discussed in § 4.1 below.

2.5 Accuracy

As with intrusion detection in general, we face the problem of *false positives* (non-backdoor connections erroneously flagged as backdoors) and *false negatives* (backdoor connections the monitor fails to detect). The former can make the detection algorithm unusable, because it becomes impossible (or at least too tedious) to examine all of the alerts manually, and attackers can exploit the latter to evade the monitor.

We would of course like to have both the false positive rate and the false negative rate be as low as possible. But particularly for those of our algorithms that are based on overall traffic characteristics rather than sharp signatures, we frequently will have to choose tradeoffs between the two.

2.6 Responsiveness

Another important design parameter is the responsiveness of the detection algorithm. That is, after a backdoor connection starts, how long does it take for the monitor to detect the backdoor? Clearly, it is desirable to detect backdoors as quickly as possible, to enable taking additional actions such as recording related traffic or shutting down the connection. However, in many cases waiting longer allows the monitor to gather more information and consequently can detect backdoors more accurately, resulting in a tradeoff of responsiveness versus accuracy.

Another consideration related to responsiveness concerns the system resources consumed by the detection algorithm. If we want to detect backdoors quickly, then we must take care not to require more resources than the monitor can devote to detection over a short time period. On the other hand, if off-line analysis is sufficient, then we can use more resource-intensive algorithms.

3 A General Algorithm for Detecting Interactive Backdoors

In this section we present a general algorithm for detecting interactive backdoors based on keystroke characteristics. The algorithm incorporates three types of characteristics: directionality, packet sizes, and packet interarrival times. We also find we need to exclude excessively short flows (common in our traces due to the use of scanning by automated monitoring software), which do not provide enough traffic to analyze soundly. The criterion we use is to skip analysis of any flows comprised of fewer than 8 packets or lasting less than 2 seconds, where a flow is one direction of a bidirectional TCP connection.

3.1 Exploiting connection directionality

As noted above, an interactive connection is most likely initiated by the client, unless the server has some callback mechanism. Therefore, when looking for keystrokes we need only consider traffic sent by the initiator of a connection. However, if the monitor doesn't see the establishment of the connection, that is, the connection is a *partial* connection, there is no way to tell who is the actual initiator. In this case, we must consider both flows.

If we are monitoring an access link and are only interested in detecting backdoors within the local site, we can further exploit the connection directionality and ignore all outbound flows, even if the connection is partial.

3.2 Exploiting packet length characteristics

3.2.1 The size of keystroke packets

Keystroke packets are likely to be very small, even if sent in line mode, because most commands are short. To verify this assumption, we analyzed several Internet traffic traces with a total of 2.1 million Telnet and Rlogin client data packets. Of these, 79% carried a single byte, 97% carried 3 bytes or less, and 99.7% carried 20 bytes or less.

For a trace of SSH 1.x and 2.x connections (very heavily skewed towards 1.x), we found that 28% of the 150 K client data packets had length 20 or less. (Note that those SSH connections with predominantly big packets are likely to be file transfers.)

Consequently, we use 20 bytes as our cutoff for "small" packets.

3.2.2 Characterizing the frequency of small packets

Since most keystroke packets are quite small, we can exclude those connections that don't have enough small packets. More specifically, we can devise a metric to measure the frequency of small packets in a connection, which we then use to determine whether we should exclude the connection.

The simplest metric is the ratio of the number of small packets over the total number of packets, for a suitable definition of "small packet," which per the previous section we define as 20 bytes or less of payload. Unfortunately, this metric doesn't work well in practice. Although, as stated in the previous section, over 99.7% of keystrokes are very small, such statistics are based on a large number of connections. For a specific connection, we find that the ratio can be as low as 30–40%. Consequently, in order to prevent frequent false negatives, we have to choose a conservative threshold as low as 20–30%. But with such a low threshold, the metrics have little discriminating power and can introduce too many

false positives.

To avoid such problems, we devised a metric Γ, defined in terms of S, the number of small packets, N, the total number of packets, and G, the number of gaps between small packets. A gap occurs any time two small packets are separated by at least one large packet. We then evaluate:

$$\Gamma = \frac{S - G - 1}{N}.$$

The intuition behind Γ is that consecutive small packets are strong indicators that a connection has interactive traffic. If the small packets are all spread throughout a connection, then we will have $G = S - 1$, so $\Gamma = 0$. If they are all grouped together, then $G = 0$ and Γ will reflect the relative proportion of small packets in the trace.

In our final algorithm, we set the threshold to $\Gamma = 0.2$.

3.3 Exploiting timing characteristics

As mentioned above, keystroke interarrival times come in a striking Pareto distribution, exhibiting a very broad range [PF95]. We can then exploit the tendency of machine-driven, non-interactive traffic to send packets back-to-back, with a very short interval between them, to discriminate non-interactive traffic from interactive. We do so by examining each pair of back-to-back small packet arrivals and computing the ratio α of how many of these interarrival times fall within the range 10 msec through 2 sec. (We need to take care not to include retransmitted packets in this computation.) The upper bound of 2 sec is fairly arbitrary; using 100 sec does not appreciably change the performance.

We then define a metric α to quantify how often the interarrival between two consecutive small packets falls in this range. In our final algorithm, we set the threshold to $\alpha = 0.2$.

It might appear that the criteria of $\Gamma = 0.2$ and $\alpha = 0.2$ are too lax, and singularly, they are; but jointly, they prove highly effective, as we show in § 5.7.

3.4 Making the algorithm run in real-time

In this section we discuss two considerations in using the algorithm in real-time. First, we observe that we can reduce the packet capture load a great deal by filtering on the data payload length of the packets to only capture small packets. `tcpdump` [JLM91] doesn't actually have an easy way to specify a particular range of payload sizes, but the following will filter out all packets with more than 20 bytes of payload:

```
# (packet length -
#  ip header length -
#  tcp header length) <= 20.
# That is, data length <= 20.
(ip[2:2] - ((ip[0]&0x0f)<<2) -
 (tcp[12]>>2)) <= 20
```

where the bit-shifting is required to extract the IP and TCP header lengths, which can be variable length due to the presence of IP or TCP options.

Introducing filtering does not affect the evaluation of α for a flow, since α is only computed for packets that are consecutive in the TCP sequence space (§ 3.3). However, we must take care when evaluating Γ, since now that we only see small packets, we can't accurately tell the total number of packets N transmitted by a given flow. To solve this problem, whenever we see a gap in the sequence number, we estimate the number of missing large packets in the gap as $\lceil gap/LARGE\_PKT\_SIZE \rceil$, where LARGE_PKT_SIZE is a guess at the most common size for full-sized packets. This size varies with path characteristics such as the Maximum Transmission Unit, and also depends on the particular TCP implementation, but as a rough approximation we simply use LARGE_PKT_SIZE = 500.

The other consideration for real-time detection concerns how quickly the algorithm can determine it has found a backdoor. For off-line analysis, it suffices to check whether a connection has backdoor characteristics when the connection terminates (or when the trace ends), and as we have defined Γ and α above, they are in terms of statistics computed over a connection's total lifetime.

The simplest way to adapt the algorithm to run in real time is to reevaluate Γ and α on each incoming packet. Alternatively, we can have a timer for each connection and test the connection whenever the timer goes off. Unfortunately, neither approach works well in practice. The major problem is that when we classify a connection as a non-backdoor connection, we can't just ignore the connection later on, because it's hard to tell whether the connection is indeed a non-backdoor connection, or instead actually a backdoor connection with a preamble that has non-backdoor characteristics (such as the Telnet option negotiations that precede a Telnet login dialog). Consequently, we have to keep re-testing each non-backdoor connection, which is clearly very expensive.

We address this problem by exponentially backing off the reevaluation timer. We initially choose a small timeout value for the timer (30 seconds). Subsequently, whenever a connection appears to be a non-backdoor, we increase the timeout value by a factor of 1.5, which spreads the computational load over the lifetime of the connection.

4 Special-Purpose Detection Algorithms

In this section we explore algorithms that look for signatures reflecting the use of particular protocols. If we then find servers for those protocols running on ports other than their standard ones, such instances may indicate the presence of a backdoor.

Compared to the general-purpose detection algorithm, special-purpose algorithms can better benefit from protocol-specific information, and hence are likely to be more accurate or more efficient. On the other hand, relying on protocol-specific information can make the algorithm susceptible to evasion, if the attacker can perturb the signature.

There are two major applications for special-purpose detection algorithms. First, they can be used as baseline algorithms to evaluate the performance of the general-purpose algorithm described in § 3, allowing us to understand how much performance we lose by making the algorithm more general (and hence more difficult to evade). Second, the special-purpose algorithms themselves can be used either individually or in combination with the general-purpose algorithm to detect backdoors.

In the rest of this section, we introduce 15 algorithms for detecting various interactive protocols and the like. Based on different design purposes, we can divide these algorithms into the following two classes:

- Optimal algorithms are designed to identify backdoors as accurately as possible, without worrying about efficiency. Such algorithms are intended for use as baseline algorithms and for off-line analysis.
- Efficient algorithms incorporate protocol-specific filtering mechanisms into the optimal algorithms to reduce their expense, at the cost of a degree of accuracy. The tradeoff here varies a great deal—sometimes it is even possible to use a simple packet filter to achieve accuracy in the same league as for much more expensive algorithms (see § 4.1 below)—and the gain is algorithms efficient enough to use for real-time detection.

Table 1 summarizes the algorithms discussed in the rest of this section.

Backdoor type	Optimal algorithm	Efficient algorithm
SSH	**ssh-sig, ssh-len**	**ssh-sig-filter**
Rlogin	**rlogin-sig**	**rlogin-sig-filter**
Telnet	**telnet-sig**	**telnet-sig-filter**
FTP	**ftp-sig**	**ftp-sig-filter**
Root prompt	**root-sig**	**root-sig-filter**
Napster	**napster-sig**	**napster-sig-filter**
Gnutella	**gnutella-sig**	**gnutella-sig-filter**

Table 1: Summary of the special-purpose backdoor detection algorithms.

4.1 SSH

Secure Shell (SSH) encrypts transmitted content with strong cryptography. It is increasingly used for both interactive and bulk transfer traffic. While all in all its deployment represents a major advance for Internet security, it presents significant difficulties for content-based intrusion detection precisely because it renders the monitor blind to the specifics of each connection. It is thus particularly attractive for backdoor use.

Our first algorithm for detecting SSH, **ssh-sig**, uses the SSH version string as the signature for SSH. When an SSH connection has been established, both sides send an identifying string of the form "SSH-protoversion-softwareversion comments", followed by carriage-return and newline (ASCII 13 and 10, respectively) [YKSRL99]. The maximum length of the string is 255 characters, including the carriage-return/newline. Version strings contain only printable characters, not including space or "-".

Currently, the SSH protocol version is either "1.x" or "2.x". Therefore, it suffices for **ssh-sig** to look for text "SSH-1." or "SSH-2." at the beginning of the first data packet sent in each direction of a connection.

We can replace **ssh-sig** with the following tcpdump filter (denoted as **ssh-sig-filter**) for very efficient detection:

```
# 1st 4 bytes are 'SSH-' and
# bytes 5 and 6 are '1.' or '2.'
```

```
tcp[(tcp[12]>>2):4] = 0x5353482D and
(tcp[((tcp[12]>>2)+4):2] = 0x312E or
 tcp[((tcp[12]>>2)+4):2] = 0x322E)
```

Our second detection algorithm, **ssh-len**, uses an implicit signature, the packet length, to detect SSH sessions. According to the SSH specification, SSH 1.x will (in the absence of TCP repacketization) generate packet payload sizes of the form $8k + 4$, that is, 4 more than a multiple of 8. SSH 2.x will generate payload sizes of length at least 16, and also a multiple of the cipher block size, which is a multiple of 8 for all of the ciphers of which we are aware. Therefore, for SSH, either most packets will have length $8k + 4$, or most will have length $8k$. One deviation occurs with the initial version exchange, which does not conform with these rules.

In light of this pattern, **ssh-len** detects SSH as follows:

1. First test for an interactive connection using the timing-based algorithm (§ 3). If it is interactive, go to the next step, otherwise stop.
2. If the proportion of packets with length $8k+4$ or the number of packets with length $8k$ exceeds a threshold, classify the connection as SSH.

We need to be careful when choosing the threshold, because packet retransmission and fragmentation can sometimes distort such characteristics. In our current implementation, we set the threshold to 75%.

4.2 Rlogin

Upon connection establishment, an Rlogin client sends four NUL-terminated strings to the server in the following format [Ka91]:

<NUL>
client-user-name<NUL>
server-user-name<NUL>
terminal-type/speed<NUL>

The server then returns a zero byte (NUL) to indicate that it has received these strings and is now in data transfer mode. Algorithm **rlogin-sig** attempts to detect Rlogin sessions using this negotiation as a signature. It first applies the following analysis to a connection:

- For the flow towards the initiator of a connection, check if the first byte is a NUL.
- For the flow sent by the initiator, keep testing each byte until one of the following events happens:
  - A gap in sequence number occurs;
  - four NUL's have been seen;
  - an empty string or a non-7-bit-ASCII byte is seen; or
  - the number of bytes we examined reaches a maximum bound (128 in the current algorithm).

If the above terminates by finding four NUL's, then we check to see whether the flow in the other direction begins with a non-NUL byte, or whether we found any empty strings or non-7-bit-ASCII bytes. If neither of these last two hold, then the connection is classified as an Rlogin connection.

We can combine **rlogin-sig** with the following tcpdump filter, resulting in a more efficient algorithm **rlogin-sig-filter**:

```
# last byte is 0 and data len != 0 and
# data length <= 128
(tcp[(ip[2:2]-((ip[0]&0x0f)<<2))-1]=0)
and ((ip[2:2]-((ip[0]&0x0f)<<2)-
     (tcp[12]>>2)) != 0)
and ((ip[2:2]-((ip[0]&0x0f)<<2)-
     (tcp[12]>>2)) <= 128)
```

Note that **rlogin-sig** tests for whether the *last* byte in the packet is NUL, rather than the first byte. This is necessary because we find that clients tend to send their first NUL in its own packet, and the remainder of the prolog information in a second packet.

4.3 Telnet

The Telnet protocol [PR83a] includes a quite general mechanism for negotiating options [PR83b]. Since most Telnet sessions begin with a series of option negotiations, we can attempt to detect these, which have a distinct pattern, taking one of the following four 3-byte formats:

IAC WILL *option-code*
IAC WON'T *option-code*
IAC DO *option-code*
IAC DON'T *option-code*

The code values for WILL, WON'T, DO, DON'T, and IAC are 251, 252, 253, 254, and 255 respectively. Note that some options have parameters, and so can be longer than the above three bytes.

telnet-sig tests the first two bytes of each incoming packet to see if they match the beginning of any of the above. If a connection doesn't involve any option negotiation, we classify it as a non-Telnet connection. Otherwise, we test the following additional conditions:

- At least 75% of the bytes are 7-bit-ASCII.
- At least 50% of the lines are not longer than 80 bytes.

These aid in weeding out binary traffic that happens to match the option patterns above.

We can combine the following packet filter with **telnet-sig** to form a more efficient algorithm, **telnet-sig-filter**:

```
# 1st byte is <IAC> (0xff),
# 2nd byte is <251> - <254>
(tcp[(tcp[12]>>2):2] > 0xfffa) and
(tcp[(tcp[12]>>2):2] < 0xffff)
```

4.4 FTP

In this section we look at a somewhat different form of interactive protocol, the user control portion of the FTP file transfer protocol [PR85]. FTP is a request/reply protocol in which requests are sent in single, usually short, lines of ASCII text, and replies have a similar structure, but can be longer and multi-line. Some FTP requests are sent in response to user activity, and accordingly have interactive-like timing. Others are generated mechanically by the FTP client, and arrive closely spaced.

Replies sent by FTP servers start with a status code (a number), followed by any accompanying text. For a day's worth of FTP activity between the Lawrence Berkeley National Laboratory and the rest of the Internet (7,229 connections), the distribution of the code in the first reply returned by the server is: code 220 ("ready for new user") seen 6,685 times; code 421 ("service not available") seen 535 times; code 226 ("closing data connection") seen 7 times; codes 426 ("connection closed") and 200 ("command okay") each seen once; no other codes seen.

Of these, if we miss a server that returns 421 we haven't actually missed anything significant, since the service is not available. All that really matters is detecting 220, though we can include 421, too, without too much extra effort.

For FTP server replies, the fourth byte is either a blank or a hyphen, the latter indicating a multi-line reply. Therefore, the **ftp-sig** algorithm looks in the first four bytes for either 220 or 421, followed by either a blank or a hyphen, as a signature for an FTP connection.

We can also compose **ftp-sig-filter**:

```
# 1st three bytes are '220',
# 4th byte is blank or hyphen
tcp[(tcp[12]>>2):4] = 0x3232302d or
tcp[(tcp[12]>>2):4] = 0x32323020
```

with a similar filter for 421.

One difficulty with this approach is that the same sort of status codes are used by the popular SMTP mail transfer protocol [Po82]. Code 220 corresponds to "service ready" and 421 to "service not available," just as it does for FTP. This means that our algorithms for detecting FTP backdoors should work just as well for SMTP backdoors (which can actually be beneficial), which in § 5.5 we explore further.

4.5 Root Backdoor

From operational experience we have found that one particular type of backdoor installed by attackers is a Unix root shell, and the connection to it may not involve any Telnet option negotiation. For these, often the server starts by sending a packet with a payload of exactly two bytes: "#<*blank*>", which corresponds to one of the forms of a Unix root shell prompt. This gives us a simple algorithm, **root-sig**, which attempts to detect root backdoors by looking for the two bytes in the first packet sent by the server side of a connection, and the corresponding **root-sig-filter**:

```
# look for '# ' in a packet with
# exactly 2 bytes of payload
tcp[(tcp[12]>>2):2] = 0x2320 and
(ip[2:2] - ((ip[0]&0x0f)<<2) -
(tcp[12]>>2)) == 2
```

which, given its conceptual simplicity, works surprisingly well (see § 5.6 below).

4.6 Napster

Napster is a distributed system by which users can share copies of music that has been digitized in MP3 format [NA99]. Users run a client that connects to `napster.com` servers for purposes of publishing the MP3's that the user has made available to the public, and for searching for particular MP3's available elsewhere in the distributed database. The server redirects the client to other clients that have the desired MP3 available, and the client then makes a direct connection to the source of the MP3, bypassing the server at this point.

Napster has proven controversial because often the MP3 trading is in violation of copyright laws, and also because MP3's tend to be large files, so the enthusiasm of a site's Napster users can consume considerable resources [NA00, Ha00]. Therefore, sites make efforts to control Napster traffic, for example by removing connectivity to the `napster.com` servers. Napster users have taken counter-measures to circumvent such blocking [We00], including configuring Napster servers to use non-standard ports for their communications. Open-source Napster clients are also available [GN99, ON00a], which will aid Napster users in modifying the client's behavior to better circumvent detection.

Detecting Napster traffic is thus in many ways similar to detecting other backdoors, even though in this case the traffic does not reflect a security access violation, but rather a policy violation (authorization rather than authentication).

We focused on the problem of detecting the communication directly between Napster clients (used to transfer the actual MP3's). One thought was to develop a generic MP3 detector, though our preliminary work on this has shown the problem to be somewhat difficult, as the format has a short, binary header that does not suggest a simple, distinct pattern to look for [Bo00].

The Napster client communication, however, has a quite distinctive signature [ON00b]. The communication begins with the string `SEND` or `GET` followed immediately by the name of the item (no intervening whitespace). Furthermore, we have found that the `SEND` or `GET` directive is sent by the Napster client in its own packet,[2] so our current version of **napster-sig** simply looks for either of these strings sent in their own packet and occurring at the beginning of a connection. **napster-sig-filter** does the same, but without the beginning-of-a-connection context:

```
# look for "SEND" or "GET" in a
# packet by itself (so payload of
# 4 or 3 bytes, respectively)
((ip[2:2] - ((ip[0]&0x0f)<<2) -
  (tcp[12]>>2)) = 4 and
 tcp[(tcp[12]>>2):4] = 0x53454e44) or
((ip[2:2] - ((ip[0]&0x0f)<<2) -
  (tcp[12]>>2)) = 3 and
 tcp[(tcp[12]>>2):2] = 0x4745 and
 tcp[(tcp[12]>>2)+2]=0x54)
```

4.7 Gnutella

Gnutella is a distribution system similar in spirit to Napster [GN00]. Its distinctive features are that it is fully open source, it can be used to exchange arbitrary files and not just MP3's (although there are now Napster add-ons for doing this, too), and it has no centralized component-—Gnutella clients simply need to know the name of another Gnutella client and they can participate in the distribution network. Consequently, Gnutella is likely to prove harder for sites to control than Napster.

In its current form, however, Gnutella is very easy to detect. Each Gnutella session begins with the connecting client transmitting:

`GNUTELLA CONNECT/<`*version*`><`*NL*`><`*NL*`>`

and receiving in reply:

`GNUTELLA OK<`*NL*`><`*NL*`>`

where <*NL*> is the newline character (ASCII 10).

Accordingly, **gnutella-sig** looks for the string "`GNUTELLA<`*blank*`>`" at the beginning of a connection.

The corresponding **gnutella-sig-filter** is:

```
# look for "GNUTELLA " as first
# 9 characters of payload
tcp[(tcp[12]>>2):4] = 0x474e5554 and
tcp[(4+(tcp[12]>>2)):4] = 0x454c4c41
and tcp[8+(tcp[12]>>2)] = 0x20
```

[2] Clearly, this is very easy for the Napster client to change, and the corresponding change to make to our detector is looking for the absence of whitespace following the directive, which will address mistaking Napster `GET`'s for those used by HTTP.

5 Performance evaluation

In this section we evaluate the algorithms developed in § 3 and § 4. The evaluations were done by adding implementations of the algorithms to the Bro intrusion detection system [Pa98].

Our general framework for evaluation is as follows. To assess an algorithm's accuracy, we first run it against known interactive traffic of the particular type it is supposed to detect (Telnet, Rlogin, SSH; or, for the general algorithm, a combination of Telnet and Rlogin, since SSH traffic is sometimes bulk-transfer) and analyze how often it fails to flag a connection in the trace as interactive. This evaluates the *false negative* rate. We then run the algorithm against packet traces of a site's Internet traffic (these have high-volume protocols such as HTTP, NFS, and X11 removed, because otherwise we could not capture the traces reliably) to see which connections they mark as interactive, and then manually assess whether the connection does indeed appear to be interactive. This evaluates the *false positive* rate.

Note, we do not assess the Napster and Gnutella detectors, as the traces we use here were captured before those applications existed. However, our informal assessment based on correlating traffic to known Napster and Gnutella ports and services is that they work very well.

5.1 Trace description

We used four traces to evaluate the performance of the algorithms:

- `ssh.trace` (194MB, 380K packets, 905 connections), a half-hour snapshot of all the SSH connections seen late at night on the Internet access link (DMZ) of the University of California at Berkeley (UCB).

- `lbnl.mix1.trace` (54MB, 134K packets, 4.6K connections) and `lbnl.mix2.trace` (421MB, 863K packets, 14.7K connections). Each trace contains one hour of aggregate traffic collected at the DMZ of the Lawrence Berkeley National Laboratory (LBNL), the first in the middle of the night, the second in the middle of the afternoon. The traces have had high volume protocols (HTTP, SSH, NFS, X11, NNTP, FTP data) filtered out.

  Note that we might well apply such filtering for operational use, too, deciding to trade off missing backdoors on those ports for the reduced packet capture load.

- `lbnl.inter.trace` (389MB, 3.5M packets, 5.5K connections), one day's worth of Telnet and Rlogin traffic collected at LBNL.

5.2 Performance of SSH algorithms

We ran **ssh-sig** on trace `ssh.trace` to evaluate its false negative ratio. Clearly, **ssh-sig** only works when the beginning of a connection is present. Altogether, there are 546 complete SSH connections in `ssh.trace`, none of which is missed by **ssh-sig**. This demonstrates that the false negative ratio of **ssh-sig** is extremely low, which is to be expected since the presence of the signature is required by the specification.

We then ran **ssh-sig** on `lbnl.mix1.trace`, `lbnl.mix2.trace` and `lbnl.inter.trace` to evaluate its false positive ratio. Among the 16,938 complete non-SSH connections, none is mis-classified as SSH by **ssh-sig**. Therefore, the false positive ratio of **ssh-sig** is close to 0.

ssh-sig-filter has exactly the same good performance on the traces we have, which is not surprising, as the only apparent opportunity for error is unusual packetization splitting the SSH version text across multiple packets. In addition, the filtering gain is tremendous, because only those packets that contain the SSH version string need to be further processed. For `ssh.trace`, the algorithm needs only inspect 111 KB of packets rather than the 194 MB present in the entire trace.

The major limitation of **ssh-sig** and **ssh-sig-filter** is that they only work when the beginning of an SSH connection is present.

Since SSH can be used for both interactive traffic and bulk transfer, it is difficult to soundly evaluate the false negative ratio of **ssh-len**, which is designed to detect *interactive* SSH backdoors. Consequently, we only evaluate the false positive ratio here.

Again, we ran **ssh-len** on the three traces without ssh connections: `lbnl.mix1.trace`, `lbnl.mix2.trace` and `lbnl.inter.trace`. Among the 16,938 non-SSH connections, only 5 are classified as SSH by **ssh-len**, yielding a very low false positive rate.

Compared with **ssh-sig** and **ssh-sig-filter**, **ssh-len** does not require the presence of the beginning of a connection. However, it is less robust for SSH 1.x over highly lossy links, where two SSH blocks of length $8k+4$ could be coalesced due to packet retransmission, resulting in a single packet of $8(k_1 + k_2 + 1)$ bytes. Consequently, we only use **ssh-len** when the beginning of a connection is missing.

5.3 Performance of Rlogin algorithms

Altogether there are 175 complete Rlogin connections in the traces, none of which is missed by **rlogin-sig**.

We begin with evaluating the false positive ratio of **rlogin-sig**. In the four traces, altogether there are 17,306 non-rlogin connections, none of which is mis-classified as an Rlogin connection. This means **rlogin-sig** also has an extremely low false positive ratio.

After adding filtering into **rlogin-sig**, we found that the false negative ratio remains the same (0/175). Meanwhile, the increase in the false positive ratio is marginal: altogether there are 4 out of 17,306 non-Rlogin connections that are mis-classified as Rlogin connections by **rlogin-sig-filter**.

The filtering gain of **rlogin-sig-filter** is significant. Among the 1 GB data we have in the four traces, only 16 MB data needs to be processed by **rlogin-sig**.

The major limitation of **rlogin-sig** and **rlogin-sig-filter** is similar to **ssh-sig**—they only work when the beginning of a connection is seen by the monitor.

5.4 Performance of Telnet algorithms

Again, we first evaluate the false negative ratio of algorithm **telnet-sig**. Unfortunately, it turns out that many Telnet connections in our traces are very short. For such short connections, **telnet-sig** fails because the connections do not include option negotiations. On the other hand, if a connection is that short, even if it is indeed a backdoor, it is not likely to cause significant damage.

To make the evaluation meaningful, we only consider those connections satisfying:

- the client sends at least two lines of data;
- the server sends at least one line of data; and
- the duration of the connection is at least 1 second.

After eliminating connections not satisfying these requirements, 1,526 Telnet connections remain, 18 of which are missed by **telnet-sig**. Further inspection shows that 17 out of the 18 involve the same public library catalog server, which performs passwordless logins without any option negotiation.

We further find that of the 12,708 non-Telnet connections in the traces, none is mis-classified as Telnet connections. This demonstrates that **telnet-sig** has a very low false positive ratio.

After adding filtering into **telnet-sig** to form algorithm **telnet-sig-filter**, the false positive and false negative ratios are unaffected for the traces we have studied. The filtering gain, however, is significant: **telnet-sig-filter** has to process less than 1.5 MB out of over 1 GB of packet data.

The major limitation of **telnet-sig** and **telnet-sig-filter** is similar to **ssh-sig** and **rlogin-sig**—they only work when the connection as seen by the monitor includes option negotiations, which tends to only occur at the beginning of a connection.

5.5 Performance of FTP algorithms

As noted in § 4.4, our FTP detection algorithm will also detect SMTP, so here we note this limitation and then treat the two protocols together.

We have altogether 5,629 FTP/SMTP sessions in which the server sent at least 4 bytes of data. Of these, 29 are missed by **ftp-sig**. Further inspection shows that these connections are almost all partial connections for which the initial dialog (which is far and away the most likely place for our signature to trigger) is missing. This demonstrates that **ftp-sig** has a low false negative ratio.

Among 20,135 non-FTP/SMTP connections, only one is classified as FTP/SMTP. Further inspection shows that this is actually an FTP server running via WinSock—so there is no false positive after all!

After adding filtering, **ftp-sig-filter** enjoys the same accuracy, as well as a terrific filtering gain: only 1.2 MB out of over 1 GB data need be processed by **ftp-sig-filter**.

Again, the limitation for **ftp-sig** and **ftp-sig-filter** is that,

except for rare exceptions, they only work when the beginning of a connection is seen by the monitor.

5.6 Root shell algorithms

As far as we can tell, our traces do not include any root shells, so we cannot soundly evaluate the performance of **root-sig** and **root-sig-filter**. But see the next section for preliminary experiences indicating that they (**root-sig-filter**, in particular) are quite powerful.

5.7 Performance of the general detection algorithm

To assess the false negative ratio of the algorithm, we ran it on trace `lbnl.inter.trace`, which consists only of Telnet and Rlogin connections. Among the 150 complete Rlogin connections, 26 are missed by the algorithm. Further inspection shows that 23 are excessively short (less than 2 seconds in duration, or only one command executed), and the other 3 are user login failures. Among all 1,450 Telnet connections that are not excessively short, 22 are missed by the timing-based algorithm. Therefore, the false negative ratio is at least comparable to **telnet-sig**. Further inspection shows that the algorithm found all 18 connections missed by the **telnet-sig**, but 22 connections detected by **telnet-sig** are missed by the timing-based algorithm.

To evaluate the false positive ratio of the algorithm, we ran the algorithm on `lbnl.mix1.trace` and `lbnl.mix2.trace` with all the Telnet/Rlogin/FTP/SSH/SMTP connections filtered out. Among over 12,000 connections, the timing-based algorithm reported 57 backdoors. Further inspection shows that 45 are IMAP [Cr94] and POP [MR96] mail servers used interactively, and therefore are not in fact false positives.[3]

5.8 Experience with production use

We only recently begun operational deployment of the backdoor detection algorithms for production use on the LBNL DMZ. One of the most surprising (and, in retrospect, obvious) findings has been the large number of legitimate backdoors.

[3]The algorithm has also detected interactive SMTP sessions, nominally a non-interactive protocol.

For example, when analyzing 20 minutes of traffic from the UCB DMZ (comprising 4.9 GB of data after filtering out the high volume traffic), the protocol-specific algorithms report 334 backdoors on non-standard ports. Of these, 326 are FTP servers on non-standard ports, 7 are interactive games, and the remaining one is a library card catalog server. In contrast, the timing-based algorithm reports 220 backdoors. From visual inspections of 75 of these, we found: 17 are interactive AOL sessions, 19 are interactive games, 14 are chat sessions, 3 are card catalog servers, 7 are FTP sessions, and we were unable to identify the other 15.

Running on the live traffic stream, the SSH detection algorithms have turned up SSH servers running on port 80 (nominally HTTP—the server ran on that port to provide tunneling through firewalls); port 110 (nominally POP); port 32 (used to run an older version of SSH than the one on port 22, due to compatibility problems); ports 44320–44327 (a NAT server with SSH access to the collection of hosts behind it via a number of different ports); as well as a host of variants of 22 (222, 922, 2222, ...).

For production use it is unsafe to filter out the high-volume protocols. Running the signature-based tcpdump filters on full traffic streams does not present any performance problems when using a kernel-based packet filter, as the filters are highly selective. For the other protocol-specific detectors, it appears we can also run them on good-sized full traffic streams, as running all of them against a 10 GB trace only takes about 20 CPU minutes on a 400 MHz Pentium II.

We run all of the protocol-specific detectors daily against traces of LBNL traffic other than the high-volume ports. (We will shortly be configuring our monitor to run them in real-time.) We currently run with a set of five filters to remove legitimate backdoors: the NAT front-end mentioned above; two hosts that run a document upload service that triggers `ftp-sig` (the protocol is not FTP or SMTP, but has a similar structure); a host that runs a service on TCP port 497 that involves an exchange that looks like Telnet option negotiation (but isn't); and a popular FTP server that sometimes serves files with binary data that looks like embedded Telnet options.

The Napster and Gnutella detectors have become important tools in enforcing LBNL's appropriate use policy, and, for example, have detected a remote Napster server running on port 21 (FTP) in an apparent attempt to hide or circumvent a firewall.

The root backdoor filter, **root-sig-filter**, has uncovered root backdoors running on UCB traffic. However, these

have not been in the form originally intended (in which the connection begins directly with "#<*blank*>"), which we know from experience are a rare, albeit striking, signature. Instead, because the filter version of the algorithm detects "#<*blank*>" *anywhere* in a connection, providing it is sent as a prompt (by itself with no newline), **root-sig-filter** is quite powerful at detecting both some transitions to root via the Unix *su* command, and sessions for which the prompt seen after the login prolog is indeed "#<*blank*>".

Part of the appeal of **root-sig-filter** is that it generates very few candidate connections, so even though its false hit rate on general traffic is fairly high, the connections it flags are not burdensome to check, and it is an exceptionally cheap algorithm in terms of computation.

We do not yet run the general algorithm operationally. As discussed above, it detects large numbers of interactive services, requiring time-consuming effort contacting the managers for the various machines to determine that in fact the backdoors are legitimate. But the potential of the approach seems clear already.

6 Summary

The problem of finding a backdoor connection in a flood of otherwise legitimate network traffic initially appears daunting. But because interactive traffic has characteristics quite different from most machine-driven traffic (smaller packet sizes, longer idle periods), it is possible to search efficiently for such traffic. We have presented a general algorithm for doing so, and also protocol-specific algorithms that look for signatures particular to different protocols, both of which we implemented in the Bro intrusion detection system.

One unexpected benefit of developing the protocol-specific algorithms was to realize how it is frequently possible to fingerprint a particular application protocol by unique or nearly unique text it includes. This lead to the developement of successful algorithms for Napster and Gnutella, which can be important to detect given that their use sometimes violates a site's policy, and that their users often attempt to evade detection.

The algorithms are frequently amenable to prefiltering in which a stateless packet filter discards nearly all of the traffic stream before it is even considered by the algorithm. Such filtering yields major performance increases in terms of reduced CPU processing, for little or sometimes no decrease in accuracy. A related line of future work that may prove fruitful is to explore the possibility of combining the general algorithm with the protocol-specific algorithms, which is likely to yield better accuracy.

While the algorithms work very well, a major stumbling block we failed to anticipate is the large number of legitimate "backdoors" that users routinely access. These are not backdoors in the surreptitious sense, but only in the more general sense of standard protocols being run on non-standard ports. We have recently begun using the algorithms operationally, which will necessitate both the development of refined security policies addressing the many legitimate backdoors, and honing our algorithms as a mechanistic way to eliminate certain classes of benign backdoors. But even given these hurdles, we find the utility of the detection algorithms clear and compelling, and a natural next step is to now investigate their application to detecting custom backdoor protocols such as LOKI [da97] and Back Orifice [CERT98].

7 Acknowledgments

We would like to thank Ken Lindahl and Cliff Frost for their greatly appreciated help with gaining research access to UCB's traffic, and Tara Whalen and the anonymous reviewers for their feedback on the work and its presentation.

References

[Bo90] D. Borman, "Telnet Linemode Option," RFC 1184, Network Information Center, SRI International, Menlo Park, CA, Oct. 1990.

[Bo00] G. Bouvigne, "MPEG Audio Layer I/II/III frame header," http://www.mp3-tech.org/programmer/frame_header.html, 2000.

[CERT98] CERT Vulnerability Note VN-98.07, http://www.cert.org/vul_notes/VN-98.07.backorifice.html, Oct 1998.

[Cr94] M. Crispin, "Internet Message Access Protocol - Version 4," RFC 1730, Network Information Center, DDN Network Information Center, Dec. 1994.

[da97] daemon9 route@infonexus.com, "LOKI2 (the implementation)," *Phrack Magazine*, 7(51),

Sep.01, 1997. http://www.infowar.com/iwftp/phrack/Phrack51/P51-06.txt .

[DJCME92] P. Danzig, S. Jamin, R. Cáceres, D. Mitzel, and D. Estrin, "An Empirical Workload Model for Driving Wide-area TCP/IP Network Simulations," *Internetworking: Research and Experience*, 3(1), pp. 1-26, 1992.

[Gl93] V. Gligor, "A Guide to Understanding Covert Channel Analysis of Trusted Systems," NCSC-TG-030, version 1, http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-030.html, National Computer Security Center, Nov. 1993.

[GN99] Gnapster, http://www.faradic.net/jasta/gnapster.html, 1999.

[GN00] Gnutella, http://gnutella.wego.com, 2000.

[Ha00] J. Harrow, "The Consumer Internet Steamroller," *The Rapidly Changing Face of Computing*, http://www.compaq.com/rcfoc/20000417.html #_Toc480185377, April, 2000.

[JLM91] V. Jacobson, C. Leres, and S. McCanne, "tcpdump," ftp://ftp.ee.lbl.gov/tcpdump.tar.Z, 1991.

[Ka91] B. Kantor, "BSD Rlogin," RFC 1282, Network Information Center, SRI International, Menlo Park, CA, Dec. 1991.

[LWWWG98] R. Lippmann, D. Wyschogrod, S. Webster, D. Weber, and S. Gorton, "Using Bottleneck Verification to Find Novel New Attacks with a Low False Alarm Rate," Proc. Recent Advances in Intrusion Detection, Sept. 1998; http://www.zurich.ibm.com/~dac/Prog_RAID98/Talks.html#Lippmann.21 .

[MR96] J. Myers and M. Rose, "Post Office Protocol - Version 3," RFC 1939, Network Information Center, DDN Network Information Center, May 1996.

[NA99] Napster, http://www.napster.com, 1999.

[NA00] Napster (Press Room), http://www.napster.com/press.html, 2000.

[ON00b] "Napster protocol specification," http://opennap.sourceforge.net/napster.txt, June 2000.

[ON00a] OpenNap, http://opennap.sourceforge.net, 2000.

[PF95] V. Paxson and S. Floyd, "Wide-Area Traffic: The Failure of Poisson Modeling," *IEEE/ACM Transactions on Networking*, 3(3), pp. 226-244, June 1995.

[Pa98] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Proc. USENIX Security Symposium*, Jan. 1998.

[Po82] J. Postel, "Simple Mail Transfer Protocol," RFC 821, Network Information Center, SRI International, Menlo Park, CA, Aug. 1982.

[PR83a] J. Postel and J. Reynolds, "Telnet Protocol Specification," RFC 854, Network Information Center, SRI International, Menlo Park, CA, May 1983.

[PR83b] J. Postel and J. Reynolds, "Telnet Option Specifications," RFC 855, Network Information Center, SRI International, Menlo Park, CA, May 1983.

[PR85] J. Postel and J. Reynolds, "File Transfer Protocol (FTP)," RFC 959, Network Information Center, SRI International, Menlo Park, CA, Oct. 1985.

[PN98] T. Ptacek and T. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Inc., http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps, Jan. 1998.

[Ra00] M. Ranum. "RE: Bypassing firewall," mailing list firewall-wizards@nfr.net, Feb. 1, 2000.

[We00] D. Weekly, "How to get around a Napster blockade," http://david.weekly.org/code/napster-proxy.php3, 2000.

[YKSRL99] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen, "SSH Transport Layer Protocol," Internet Draft, draft-ietf-secsh-transport-07.txt, May 2000.

Detecting Stepping Stones

Yin Zhang
Department of Computer Science
Cornell University
Ithaca, NY 14853
yzhang@cs.cornell.edu

Vern Paxson*
AT&T Center for Internet Research at ICSI
International Computer Science Institute
Berkeley, CA 94704
vern@aciri.org

Abstract

One widely-used technique by which network attackers attain anonymity and complicate their apprehension is by employing *stepping stones*: they launch attacks not from their own computer but from intermediary hosts that they previously compromised. We develop an efficient algorithm for detecting stepping stones by monitoring a site's Internet access link. The algorithm is based on the distinctive characteristics (packet size, timing) of interactive traffic, and not on connection contents, and hence can be used to find stepping stones even when the traffic is encrypted. We evaluate the algorithm on large Internet access traces and find that it performs quite well. However, the success of the algorithm is tempered by the discovery that large sites have many users who routinely traverse stepping stones for a variety of legitimate reasons. Hence, stepping-stone detection also requires a significant policy component for separating allowable stepping-stone pairs from surreptitious access.

1 Introduction

A major problem with apprehending Internet attackers is the ease with which attackers can hide their identity. Consequently, attackers run little risk of detection. One widely-used technique for attaining anonymity is for an attacker to use *stepping stones*: launching attacks not from their own computer but from intermediary hosts that they previously compromised. Intruders often assemble a collection of accounts on compromised hosts, and then when conducting a new attack they log-in through a series of these hosts before finally assaulting the target. Since stepping stones are generally heterogeneous, diversely-administered hosts, it is very difficult to trace an attack back through them to its actual origin.

There are a number of benefits to detecting stepping stones: to flag suspicious activity; to maintain logs in case a break-in is subsequently detected as having come from the local site; to detect inside attackers laundering their connections through external hosts; to enforce policies regarding transit traffic; and to detect insecure combinations of legitimate connections, such as a clear-text Telnet session that exposes an SSH passphrase.

The problem of detecting stepping stones was first addressed in a ground-breaking paper by Staniford-Chen and Heberlein [SH95]. To our knowledge, other than that work, the topic has gone unaddressed in the literature. In this paper, we endeavor to systematically analyze the stepping stone detection problem and devise accurate and efficient detection algorithms. While, as with most forms of intrusion detection, with enough diligence attackers can generally evade detection [PN98], our ideal goal is to make it painfully difficult for them to do so.

The rest of the paper is organized as follows. We first examine the different tradeoffs that come up when designing a stepping stone algorithm (§ 3). We then in § 4 develop a timing-based algorithm that works surprisingly well, per the evaluation in § 5, and also evaluate two cheap context-based techniques. We conclude in § 6 with some of the remaining challenges: in particular, the need for rich monitoring policies, given our discovery that legitimate stepping stones are in fact very common; and the possibility of detecting non-interactive *relays* and *slaves*.

2 Terminology and Notation

We begin with terminology. When a person (or a program) logs into one computer, from there logs into another, and perhaps a number still more, we refer to the

*Also with the Lawrence Berkeley National Laboratory.

sequence of logins as a *connection chain* [SH95]. Any intermediate host on a connection chain is called a *stepping stone*. We call a pair of network connections a *stepping stone connection pair* if both connections are part of a connection chain.

Sometimes we will differentiate between *flow* and *connection*. A *bidirectional* connection consists of two *unidirectional* flows. We term the series of flows along each direction of a connection chain a *flow chain*.

We use the following additional notation:

- $h_1 \leftrightarrow h_2$: a bi-directional network connection between h_1 and h_2. We also use C_1, C_2, ... to denote network connections.
- $h_1 \rightarrow h_2$: a unidirectional flow from h_1 to h_2.
- $\equiv_{stepping}$ is a binary relation defined over all connections as follows: $C_1 \equiv_{stepping} C_2$ if and only if C_1 and C_2 form a stepping stone connection pair.

3 Design Space

In this section we discuss the tradeoffs of different high-level design considerations when devising algorithms to detect stepping stones. Some of the choices relate to the following observation about stepping-stone detection: intuitively, the difference between a stepping stone connection pair and a randomly picked pair of connections is that the connections in the stepping stone pair are much more likely to have some correlated traffic characteristics. Hence, a general approach for detecting stepping stones is to identify traffic characteristics that are *invariant* or at least highly correlated across stepping stone connection pairs, but not so for arbitrary pairs of connection. Some potential candidates for such invariants are the connection contents, inter-packet spacing, ON/OFF patterns of activity, traffic volume or rate, or specific combinations of these. We examine these as they arise in the subsequent discussion.

3.1 Whether to analyze connection contents

A natural approach for stepping-stone detection is to examine the contents of different connections to find those that are highly similar. Such an approach is adopted in [SH95] and proves effective. Considerable care must be taken, though, because we will not find a perfect match between two stepping stone connections. They may differ due to translations of characters such as escape sequences, or the varying presence of Telnet options [PR83b].

In addition, suppose we are monitoring connections $h_1 \leftrightarrow h_2$ and $h_2 \leftrightarrow h_3$, where h_2 is the stepping stone the attacker is using to access h_3 from h_1. If we adopt a notion of "binning" in order to group activity into different time regions (for example to compute character frequencies as done in [SH95]) then due to the lag between activity on $h_1 \leftrightarrow h_2$ and activity on $h_2 \leftrightarrow h_3$, the contents falling into each bin will match imperfectly. Furthermore, if the attacker is concurrently attacking h_4 via h_2, then the traffic on $h_1 \leftrightarrow h_2$ will be a mixture of that from $h_2 \leftrightarrow h_3$ and that from $h_2 \leftrightarrow h_4$, and neither of the latter connections' contents will show up exactly in $h_1 \leftrightarrow h_2$.

These considerations complicate content-based detection techniques. A more fundamental limitation is that content-based techniques cannot, unfortunately, work when the content is encrypted, such as due to use of SecureShell (SSH; [YKSRL99]).

The goal of our work was to see how far we could get in detecting stepping stones without relying on packet contents, because by doing so we can potentially attain algorithms that are more robust. Not relying on packet contents also yields a potentially major performance advantage, which is that we then do not need to capture entire packet contents with the packet filter, but only packet headers, considerably reducing the packet capture load. However, we also devised two cheap content-based techniques for purposes of comparison (§ 5.3), neither of which is robust, but both of which have the virtue of being very simple.

3.2 Direct vs. indirect stepping stones

Suppose h_1, h_2, h_3 is a connection chain. The *direct* stepping stone detection problem is to detect that h_2 is a stepping stone if we are observing network traffic that includes the packets belonging to $h_1 \leftrightarrow h_2$ and $h_2 \leftrightarrow h_3$. If, however, the connection chain is $h_1, h_2, \ldots, h_3, h_4$, then the *indirect* stepping stone detection problem is to detect that connections $h_1 \leftrightarrow h_2$ and $h_3 \leftrightarrow h_4$ form a stepping stone pair, given that we can observe their traffic but *not* the traffic belonging to $h_2 \ldots h_3$ (and hence there is no obvious connection between h_2 and h_3).

Detecting direct stepping stones can be simpler than detecting indirect ones because for direct ones we can often greatly reduce the number of candidates for connection pairs. On the other hand, it is much easier for attackers to elude direct stepping stone detection by simply introducing an additional hop in the stepping stone chain. Furthermore, if we can detect indirect stepping stones then we will have a considerably more flexible and robust algorithm, one which can, for example, be applied to traffic traces gathered at different places (see below).

In this paper we focus on the more general problem of detecting indirect stepping stones.

3.3 Real-time detection vs. off-line analysis

We would like to be able to detect stepping stones in real-time, so we can respond to their detection before the activity completes. Another advantage of real-time detection is that we don't have to store the data for all of the traffic, which can be voluminous. For instance, a day's worth of interactive traffic (Telnet/Rlogin) at the University of California in Berkeley on average comprises about 1 GB of storage for 20,000 connections.

Algorithms that only work using off-line analysis are still valuable, however, for situations in which retrospective detection is needed, such as when an attacked site contacts the site from which they were immediately attacked. This latter site could then consult its traffic logs and run an off-line stepping stone detection algorithm to determine from where the attacker came into their own site to launch the attack.

Since real-time algorithms generally can also be applied to off-line analysis, we focus here on the former.

3.4 Passive monitoring vs. active perturbation

Another design question is whether the monitor can only perform passive monitoring or if it can actively inject perturbing traffic to the network. Passive monitoring has the advantage that it doesn't generate additional traffic, and consequently can't disturb the normal operation of the network. On the other hand, an active monitor can be more powerful in detecting stepping stones: after the monitor finds a stepping-stone candidate, it could perturb one connection in the pair by inducing loss or delay, and then look to see whether the perturbation is echoed in the other connection. If so, then the connections are very likely correlated.

Here we focus on passive monitoring, both because of its operational simplicity, and because if we can detect stepping stones using only passive techniques, then we will have a more broadly applicable algorithm, one that works without requiring the ability to manipulate incidental traffic.

3.5 Single vs. multiple measurement points

Tracing traffic at multiple points could potentially provide more information about traffic characteristics. On the other hand, doing so complicates the problem of comparing the traffic traces, as now we must account for varying network delays and clock synchronization. In this paper, we confine ourselves to the single measurement point case, with our usual presumption being that that measurement point is on the access link between a site and the rest of the Internet.

3.6 Filtering

An important factor for the success of some forms of real-time stepping-stone detection is filtering. The more traffic that can be discarded on a per-packet basis due to patterns in the TCP/IP headers, the better, as this can greatly reduce the processing load on the monitor.

However, there is clearly a tradeoff between reduced system load and lost information. First, if a monitor detects suspicious activity in a filtered stream, often the filtering has removed sufficient accompanying context that it becomes quite difficult determining if the activity is indeed an attack. In addition, the existence of filtering criteria makes it easier for the attackers to evade detection by manipulating their traffic so that it no longer matches the filtering criteria. For example, an evasion against filtering based on packet size (see below) is to use a Telnet client modified to send a large number of do-nothing Telnet options along with each keystroke or line of input.

The main likely filtering criteria for stepping-stone detection is packet size. Keystroke packets are quite small. Even when entire lines of input are transferred using "line mode" [Bo90], packet payloads tend to be much smaller than those used for bulk-transfer protocols. Therefore, by filtering packets to only capture small packets, the monitor can significantly reduce its packet capture load (for example, by weeding out heavy bulk-transfer SSH sessions while keeping interactive ones).

3.7 Minimizing state for connection pairs

Since potentially there can be a large number of active connections seen by the monitor, it is often infeasible to keep stepping-stone state for all possible pairs of connections due to the N^2 memory requirements. Therefore we need mechanisms that allow us to only keep state for a small subset of the possible connection pairs.

One approach is to limit our analysis to only detecting direct stepping stones, but for the reasons discussed in § 3.2 above, this is unappealing. There are, however, other mechanisms that work well:

- Remove connection pairs sharing the same port on the same host. If $h_1 \leftrightarrow h_2$ and $h_2 \leftrightarrow h_3$ both use port p on host h_2, then most likely the two connections are merely using the same server on h_2, rather than h_1 accessing a server on h_2 and then from that server running a client on h_2 to access a server on h_3. Removing such connection pairs is particularly helpful when there are a large number of connections connecting to the same popular server—without such filtering, when k connections connect to the same server, we need to keep state for $\frac{k(k-1)}{2}$ connection pairs!

  Note that this mechanism is worth applying even if we also test for directionality (see below), because when the monitor analyzes already-existing connections, their directionality is not necessarily apparent.

- Remove connection pairs with inconsistent directions. Depending on the topology of the network monitoring point, we may be able to classify connections as "inbound" or "outbound." If so, then we can eliminate as connection pair candidates any pairs for which both connections are in the same direction. While these connections may in fact form a chain, if the monitoring location is a chokepoint, meaning the sole path into or out of the site, then in this case there will be another connection in the opposite direction with which we can pair either of these two connections. However, if the site has multiple ingress/egress points, then we can only safely apply such filtering if all such points are monitored and the monitors coordinate with one another.

- Remove connection pairs with inconsistent timing. If two connections are a stepping stone pair, then the "upstream" (closer to the attacker) connection should encompass the downstream connection: that is, it should start first and end last. Accordingly, we can remove from our analysis any connection pairs for which the connection that started earlier also terminates earlier.

  Note that there are two risks with this filtering. First, it may be that the upstream connection terminates slightly sooner than the downstream connection, because of details of how the different TCP shutdown handshakes occur. Second, this filtering may open up the monitor to evasion by an attacker who can force their upstream connection to terminate while leaving the downstream connection running.

3.8 Traffic patterns

We can coarsely classify network traffic as either exhibiting ON/OFF activity, or running fairly continuously. For the former, we can potentially exploit the traffic's timing structure (whether the ON/OFF patterns of two connections are similar). For the latter, we can potentially exploit traffic volume information (whether two connections flow at similar rates). In addition, even for continuous traffic, if the communication is reliable, any delays resulting from waiting to detect loss and retransmit may impose enough of an ON/OFF pattern on the traffic that we can again look for timing similarities between connections.

In this paper, we focus on traffic exhibiting ON/OFF patterns, as that is characteristic of interactive traffic, which arguably constitutes the most interesting class of stepping-stone activity.

3.9 Accuracy

As with intrusion detection in general, we face the problem of *false positives* (non-stepping-stone connections erroneously flagged as stepping stones) and *false negatives* (stepping stones the monitor fails to detect). The former can make the detection algorithm unusable, because it becomes impossible (or at least too tedious) to examine all of the alerts manually, and attackers can exploit the latter to evade the monitor.

In practice, the problem of comparing connections looking for similarities can be complicated by clock synchronization (if comparing measurements made by different monitors), propagation delays (the lag between traffic showing up on one connection and then appearing on the

other), packet loss and retransmission, and packetization variations. Moreover, an intruder can intentionally inject noise in an attempt to evade the monitor. Therefore, the detection mechanism must be highly robust if it is to avoid excessive false negatives.

3.10 Responsiveness

Another important design parameter is the responsiveness of the detection algorithm. That is, after a stepping-stone connection starts, how long does it take for the monitor to detect it? Clearly, it is desirable to detect stepping stones as quickly as possible, to enable taking additional actions such as recording related traffic or shutting down the connection. However, in many cases waiting longer allows the monitor to gather more information and consequently it can detect stepping stones more accurately, resulting in a tradeoff of responsiveness versus accuracy.

Another consideration related to responsiveness concerns the system resources consumed by the detection algorithm. If we want to detect stepping stones quickly, then we must take care not to require more resources than the monitor can devote to detection over a short time period. On the other hand, if off-line analysis is sufficient, then we can use potentially more resource-intensive algorithms.

3.11 Open vs. evasive attackers

In general, intrusion detection becomes much more difficult when the attacker actively attempts to evade detection by the monitor [PN98, Pa98]. The difference between the two can come down to the utility of relying on heuristics rather than airtight algorithms: heuristics might work well for "open" (non-evasive) attackers, but completely fail in the face of an actively evasive attacker.

While ideally any detection algorithms we develop would be resistant to evasive attackers, ensuring such robustness can sometimes be exceedingly difficult, and we proceed here on the assumption that there is utility in "raising the bar" even when a detection algorithm can be defeated by a sufficiently aggressive attacker. In particular, for timing-based algorithms such as those we develop, we would like it to be the case that the only way to defeat the algorithm is for an attacker to have to introduce large delays in their interactive sessions, so that their inconvenience is maximized. We assess our algorithm's resistance to evasion in § 4.4.

4 A Timing-Based Algorithm

In this section we develop a stepping-stone detection algorithm that works by correlating different connections based solely on timing information. As discussed in the previous section, our design is motivated in high-level terms by the basic approach of identifying invariants. Moreover, the algorithm leverages the particulars of how interactive traffic behaves. This leads to an algorithm that is very effective for detecting interactive traffic (see evaluation in § 5), and should work well for detecting other forms of traffic that exhibit clear ON/OFF patterns.

4.1 ON/OFF periods

We begin by defining ON and OFF periods. When there is no data traffic on a flow for more than T_{idle} seconds, the connection is considered to be in an *OFF period.* We consider a packet as containing data only if it carries new (non-retransmitted, non-keepalive) data in its TCP payload. When a packet with non-empty payload then appears, the flow ends its OFF period and begins an *ON period,* which lasts until the flow again goes data-idle for T_{idle} seconds.

The motivation for considering traffic as structured into ON and OFF periods comes from the strikingly distinct distribution of the spacing between user keystrokes. Studies of Internet traffic have found that keystroke interarrivals are very well described by a Pareto distribution with fixed parameters [DJCME92, PF95]. The parameters are such that the distribution exhibits *infinite variance*, which in practical terms means a very wide range of values. In particular, large values are not uncommon: about 25% of keystroke packets come 500 msec or more apart, and 15% come 1 sec or more apart (1.6% come 10 sec or more apart). Thus, interactive traffic will often have significant OFF times. We can then exploit the tendency of machine-driven, non-interactive traffic to send packets back-to-back, with a very short interval between them, to discriminate non-interactive traffic from interactive.

4.2 Timing correlation when OFF periods end

The strategy underlying the algorithm is to correlate connections based on coincidences in when connection OFF periods end, or, equivalently, when ON periods begin.

Intuitively, given two connections C_1 and C_2, if

$C_1 \equiv_{stepping} C_2$, it is very likely that C_1 and C_2 *often* leave OFF periods at *similar* times—the user presses a keystroke and it is sent along first C_1 and then shortly along C_2, or a program they have executed finishes running and produces output or they receive a new shell prompt (in which case the activity ripples from C_2 to C_1).

The inverse is also likely to be true. That is, if C_1 and C_2 often leave OFF periods at similar times, then it is likely that $C_1 \equiv_{stepping} C_2$, because there are not many other mechanisms that can lead to such coincidences. (We discuss two such mechanisms in § 5.7: periodic traffic with slightly different periods, and broadcast messages.)

By quantifying *similar* and *often*, we transform the above strategy into the following detection criteria:

1. We consider two OFF periods *correlated* if their ending times differ by $\leq \delta$, where δ is a control parameter.
2. For two connections C_1 and C_2, let OFF_1 and OFF_2 be the number of OFF periods in each, and $\text{OFF}_{1,2}$ be the number of these which are correlated. We then consider C_1 and C_2 a stepping stone connection pair if:

$$\frac{\text{OFF}_{1,2}}{\min(\text{OFF}_1, \text{OFF}_2)} \geq \gamma,$$

where γ is a control parameter, which we set to 0.3.

A benefit of this approach is that the work is done *only after significant idle periods*. For busy, non-idle connections (far and away the bulk of traffic), we do nothing other than note that they are still not idle. Related to this, we need consider only a small number of possible connection pairs at any given time, because we can ignore both those that are active and those that are idle; we need only look at those that have transitioned from idle to active, and that can't happen very often because it first requires the connection to be inactive for a significant period of time. Consequently, the algorithm does not require much state to track stepping-stone pair candidates.

Because of the very wide range of keystroke interarrival times, the algorithm is not very sensitive to the choice of T_{idle}. In our current implementation, we set $T_{\text{idle}} = 0.5$ sec. In § 5.6 we briefly discuss the effects of using other values.

Finally, because we only consider correlations of when ON periods *begin*, rather than when they *end*, we are more robust to differences in throughput capacities. For two connections $C_1 \equiv_{stepping} C_2$, if C_1's throughput capacity is significantly smaller than C_2's, then an ON period on C_2 may end sooner than on C_1 (where the echo of the same data takes longer to finish transferring); but regardless of this effect, ON periods will *start* at nearly the same time.

4.3 Refinements

The scheme outlined above is appealing because of its simplicity, but it requires some refinements to improve its accuracy. The first of these is to exploit timing *causality*, based on the following observation: if two flows F_1 and F_2 are on the same *flow chain*, then their timing correlation should have a consistent ordering. If we once observe that F_1 ends its OFF period before F_2, then it should be true that F_1 *always* ends its OFF period before F_2. Confining our analysis in this way weeds out many false pairs.

To further improve the accuracy of the algorithm, we use the number of *consecutive* coincidences in determining the frequency of coincidences, because we expect consecutive coincidences to be more likely for true stepping stones than for accidentally coinciding connections. More specifically, in addition to the test in § 4.2, to consider two connections C_1 and C_2 a stepping stone connection pair we require:

$$\text{OFF}^*_{1,2} \geq \min_{\text{csc}} \quad \text{and} \quad \frac{\text{OFF}^*_{1,2}}{\min(\text{OFF}_1, \text{OFF}_2)} \geq \gamma',$$

where $\text{OFF}^*_{1,2}$ is the number of consecutive coincidences, OFF_1 and OFF_2 are as before, and $\min_{\text{csc}}$ and γ' are new control parameters. We initially used only the first of these refinements, requiring either $\min_{\text{csc}} = 2$ or $\min_{\text{csc}} = 4$ consecutive coincidences, for direct or indirect stepping stones, respectively. This in general works very well, but we added the second requirement when we found that very long-lived connections could sometimes eventually generate consecutive coincidences just by chance. These can be eliminated by very low γ' thresholds; we use $\gamma' = 2\%$ and $\gamma' = 4\%$ for direct and indirect stepping stones, respectively.

4.4 Resistance to evasion

Since the heart of the timing algorithm is correlating idle periods in two different connections, an attacker can attempt to thwart the algorithm by avoiding introducing

any idle times to correlate; introducing spurious idle times on one of the connections not reflected in the other connection; or stretching out the latency lag between the two connections to exceed δ.

To avoid connection idle times, it will likely not suffice for the attacker to simply resolve to type quickly. Given $T_{\text{idle}} = 0.5$ sec (§ 5.6), it just takes a slight pause to think, or delay by the server in generating responses to commands, to introduce an idle time.

A mechanical means such as establishing a steady stream of traffic on one of the connections but not on the other seems like a better tactic. If the intermediary and either upstream or downstream hosts run custom software, then doing so is easy, though this somewhat complicates the attacker's use of the intermediary, as now they must install a custom server on it. Another approach would be to use a mechanism already existing in the protocol between the upstream host and the intermediary to exchange traffic that the intermediary won't propagate to the downstream host; for example, an ongoing series of Telnet option negotiations. However, as particular instances of such techniques become known, they may serve as easily-recognized *signatures* for stepping stone connections instead.

Even given the transmission of a steady stream of traffic, idle times might still appear, either accidentally, due to packet loss and retransmission lulls, or purposefully, by a site introducing occasional 500 msec delays into its interactive traffic to see whether a delay shows up in a connection besides the one deliberately perturbed. Such delays might prove difficult for an attacker to mask.

The attacker might instead attempt to introduce a large number of idle times on one connection but not on the other, so as to push the ratio of idle time coincidences below γ. This will also require running custom software on the intermediary, and, indeed, this approach and the previous one are in some sense the same, aiming to undermine the basis of the timing analysis. The natural counter to this evasion tactic is to lower γ, though this of course will require steps to limit or tolerate the ensuing additional false positives. It might also be possible to detect unusually large numbers of idle periods, though we have not characterized the patterns of multiple idle periods to assess the feasibility of doing so.

Another approach an attacker might take is to pick an intermediary for which the latency lag between the two connections is larger than δ, which we set to 80 msec in § 5.6. Doing so simply by exploiting the latency between the monitoring point and the intermediary is not likely to work well, as for most sites the latency between an internal host and a monitoring point will generally be well below 40 msec; however, if an internal host connected via a very slow link (such as a modem) is available, then that may serve. Another approach would be to run a customized server or client on the intermediary that explicitly inserts the lag of 80 msec. This approach appears a significant concern for the algorithm, and may require use of much larger values of δ, so as to render the delay highly inconvenient for the attacker (80 msec is hardly noticeable, much less inconvenient). This is a natural area for future work.

5 Performance Evaluation

In § 4 we developed a timing-based algorithm for stepping stone detection. We have implemented the algorithm in Bro, a real-time intrusion detection system [Pa98]. In this section, we evaluate its performance (in terms of false positives and false negatives) on traces of wide-area Internet traffic recorded at the DMZ access link between the global Internet and two large institutions, the Lawrence Berkeley National Laboratory (LBNL) and the University of California at Berkeley (UCB).

5.1 Traces used

We ran the timing-based algorithm on numerous Internet traces to evaluate its performance. Due to space limitations, here we confine our discussion to the results for two traces:

- `lbnl-telnet.trace` (120 MB, 1.5M packets, 3,831 connections): one day's worth of Telnet and Rlogin traffic collected at LBNL. (The traffic is more than 90% Telnet.)

- `ucb-telnet.trace` (390 MB, 5M packets, 7,319 connections): 5.5 hours' worth of Telnet and Rlogin traffic collected at UCB during the afternoon busy period.

The performance of the algorithm on other traces is comparable.

5.2 Brute force content-based algorithm

To accurately evaluate the algorithms, we first devised an off-line algorithm using brute-force content matching.

The principle behind the algorithm is that, for stepping stones, each line typed by the user is often echoed verbatim across the two connections (when the content is not encrypted). Therefore, by looking at lines in common, we can find connections with similar content. With additional manual inspection, we can identify the stepping stones.

The algorithm works as follows:

1. Extract the aggregate Telnet and Rlogin output (computer-side response), for all of the sessions in the trace, into a file.

2. For each different line in the output, count how many times it occurred (this is just *sort* | *uniq -c* in Unix).

3. Throw away all lines except those appearing exactly twice. The idea is that these are good candidates for stepping stones, in that they are lines unique to either one or at most two connections.

4. Find the connection(s) in which each of these lines appears. This is done by first building a single file listing every unique line in every connection along with the name of the connection, and then doing a database *join* operation between the lines in that file and those in the list remaining after the previous step.

   If a line appears in just one connection, throw the line away.

5. Count up how many of the only-seen-twice lines each pair of connections has in common (using the Unix *join* utility).

6. Connection pairs with 5 or more only-seen-twice lines in common are now candidates for being stepping stones.

7. Of those, discard the pair if both connections are in the same direction (both into the site or both out of the site).

8. Of the remainder, visually inspect them to see whether they are indeed stepping stones. Most are; a few are correlated due to common activities such as reading the same mail message or news article.

Clearly the methodology is not airtight, and it fails completely for encrypted traffic. But it provides a good baseline assessment of the presence of clear-text stepping stones, and detects them in a completely different way than the timing algorithm does, so it is suitable for calibration and performance evaluation.

For large traces, the requirement of 5 or more lines allows us to significantly reduce the number of connection pairs that we need to visually inspect in the end. This appears to be necessary in order to make the brute-force content matching feasible.

For small- to medium-sized traces, we also inspect the ones with 2, 3, or 4 lines in common. Sometimes we did indeed find stepping stones that were missed if we required 5 lines in common. But in most cases, these stepping stones were exceedingly short in terms of bytes transferred.

5.3 Simple content-based algorithms

For purposes of comparison, we devised two simple content-based algorithms. Both are based on the notion that if we can find text in an interactive login C_1 unique to that login, then if that text also occurs in C_2, then we have strong evidence that C_1 and C_2 are related.

The problem then is to find such instances of unique text. Clearly, virtually all login sessions are unique in some fashion, but the difficulty is to cheaply detect exactly how.

Our first scheme relies on the fact that some Telnet clients propagate the X-Windows `DISPLAY` environment variable [Al94] so that remote X commands can locate the user's X display server. The value of `DISPLAY` should therefore be unique, because it globally identifies a particular instance of hardware.

We modified Bro to associate with each active Telnet session the value of `DISPLAY` propagated by the Telnet environment option (if any), and to flag any session that propagates the same value as an already existing session. We find, however, that this method has little power. It turns out that `DISPLAY` is only rarely propagated in Telnet sessions, and, in addition, non-unique values (such as hostnames not fully qualified, or, worse, strings like "`localhost.localdomain:0.0`") are propagated.[1]

[1] However, we *have* successfully used `DISPLAY` propagation to backtrace attackers, so recording it certainly has some utility.

Our second scheme works considerably better. The observation is that often when a new interactive session begins, the login dialog includes a status line like:

```
Last login: Fri Jun 18 12:56:58
     from hostx.y.z.com
```

The combination of the timestamp (which of course changes with each new login session) and the previous-access host (even if truncated, as occurs with some systems) leads to this line being frequently unique.

We modified Bro to search for the following regular expression in text sent from the server to the client:

```
/^([Ll]ast +(successful)? *login)/ |
/^Last interactive login/
```

We found one frequent instance of false positives. Some instances of the Finger service [Zi91] report such a "last login" as part of the user information they return. Thus, whenever two concurrent interactive sessions happened to finger the same user, they would be marked as a stepping stone pair. We were able to filter such instances out with a cheap test, however: it turns out that the Finger servers also terminate the status line with ASCII-1 ("control-A").

We refer to this scheme as "login tag", and compare its performance with that of the timing algorithm below. It works remarkably well considering its simplicity. Of course, it is not very robust, and fails completely for a large class of systems that do not generate status lines like the above, though perhaps for those a similar line can be found.

5.4 Accuracy

We first evaluate the accuracy of the algorithms in terms of their false negative ratio and false positive ratio. For `lbnl-telnet.trace`, we identified 23 stepping stone connection pairs among a total of 3,831 connections using the brute-force content matching as described above. (We inspected all connections with 2 or more lines in common, so 23 should be a very accurate estimation of the number of stepping stones.) One stepping stone is indirect (§ 3.2), the others were direct.

The timing-based detection algorithm reports 21 stepping stones, with no false positives and 2 false negatives. Both false negatives are quite short: one lasts for 15 seconds and the other lasts for 34 seconds.

For `ucb-telnet.trace`, due to the large volume of the data, for the brute-force technique we only inspected connections with 5 or more lines in common. We identified 47 stepping stones. In contrast, the timing-based algorithm detects 74 stepping stones. 5 out of the 47 stepping stones we identified using brute-force were missed by the timing algorithm. Among the 5 false negatives, 3 are very short either in terms of duration (less than 12 seconds) or in terms of the bytes typed (in one connection, the user logs in and immediately exits). We discuss the additional 32 stepping stones detected by the timing-based algorithm, but not by the brute-force technique, below.

To further assess performance, we ran both the "display" and the "login tag" schemes (§ 5.3) on `ucb-telnet.trace`. The "display" scheme reported 3 stepping stones, including one missed by the timing-based algorithm. "login tag" reported 20 stepping stones (plus one false positive, not further discussed here). Of these 20, the timing-based algorithm only missed one, which was exceedingly short—all the user did during the downstream session was to type `exit` to terminate the session. (This is also the stepping stone that was detected by the "display" algorithm but not by the timing algorithm.)

In summary, the timing-based algorithm has a low false negative ratio. To make sure that this does not come at the cost of a high false positive ratio, we visually inspected the additional 32 stepping stones reported by the timing-based algorithm for `ucb-telnet.trace` to see which were false positives.

It turns out that all of them were actual stepping stones. For example, there were a couple of stepping stones that used *ytalk*, a chat program. These fooled the brute-force content matching algorithm due to a lot of cursor motions. Another stepping stone fooled the content-matching approach because retransmitted data showed up in one of the transcripts but not the other.

Thus, we find that the timing-based algorithm is highly accurate in terms of both false positive ratio and false negative ratio, and works considerably better than the brute-force algorithm that we initially expected would be highly accurate.

5.5 Efficiency

The timing-based algorithm is fairly efficient. Under the current parameter settings, on a 400MHz Pentium II machine running FreeBSD 3.3, it takes 69 real-time seconds for `lbnl-telnet.trace`, and about 24 minutes for `ucb-telnet.trace`. The former clearly suffices for real-time detection. The latter, for a 5.5 hour trace, reflects about 10% of the CPU, and would appear likewise to suffice. Note that the relationship between the running time on the two traces is not linear in the number of packets or connections in the trace, because what instead matters is the number of *concurrent* connections, as these are what lead to overlapping ON/OFF periods that require further evaluation.

5.6 Impact of different control parameters

The proper choice of the control parameters is important for both the accuracy and the efficiency of the algorithm. We based the current choice of parameters on extensive experiments with various traffic traces, which we summarize in this section. With these settings, the algorithm performs very well in terms of both accuracy and speed across a wide range of network scenarios.

Parameter	Values
T_{idle} (sec)	0.5
δ (msec)	20, 40, 80, 120, 160
γ	15%, 30%, 45%
min_{csc}	1, 2, 4, 8, 12, 16
γ'	2% for direct stepping stones; 4% for indirect stepping stones

Table 1: Settings for different control parameters.

To assess the impact of the different control parameters, we systematically explored the portions of the parameter space on `ucb-telnet.trace`. Table 1 summarizes the different parameter settings we considered. Note that we keep the default settings for T_{idle} and γ' when exploring the parameter space, which we did to keep the size of the parameter space tractable. We chose to not vary these two parameters in particular because based on extensive experiments with various traffic traces, we have found that:

- The algorithm is fairly insensitive to the choice of T_{idle}. This is largely because, as noted in § 4.1, human keystroke interarrivals are well described by a Pareto distribution with fixed parameters. The Pareto distribution has a distinctive "heavy-tail" property, i.e., pretty much no matter what value we choose for T_{idle}, we still have an appreciable number of keystrokes to work with. However, the larger the T_{idle}, the more likely that we will miss short stepping stones. The current choice of 0.5 sec is a reasonable compromise between exceeding most round-trip times (RTTs), yet maintaining responsiveness to short-lived connections.
- Although the current choices of γ' thresholds are very low, they suffice to eliminate those very long-lived connections that eventually generate consecutive coincidences just by chance, which is the only purpose for introducing γ'.

Finally, an important point is that the goal for this assessment is determining the best parameters to use for an unaware attacker. If the attacker actively attempts to *evade* detection, then as noted in § 4.4 alternative parameters may be required even though they work less well in general. The important problem of assessing how to optimize the algorithm for this latter environment remains for future work.

We ran the detection algorithm on `ucb-telnet.trace` for each of the 75 possible combinations of the control parameters and assessed the number of false positives and false negatives. For brevity, we only report the complete results for $\gamma = 30\%$, and briefly summarize the results for $\gamma = 15\%$ and $\gamma = 45\%$.

FP/FN (γ=30%)

	min_{csc}					
δ (msec)	1	2	4	8	12	16
20	1/8	0/8	0/10	0/17	0/21	0/26
40	1/6	0/7	0/10	0/17	0/21	0/25
80	4/5	0/7	0/9	0/16	0/20	0/24
120	12/5	0/7	0/9	0/15	0/19	0/24
160	20/5	0/7	0/9	0/14	0/19	0/24

Table 2: Number of false positives (FP) and false negatives (FN) for detecting direct stepping stones when $\gamma = 30\%$.

Table 2 gives the results for detecting direct stepping stones when $\gamma = 30\%$. We make four observations. First, the number of false positives is close to 0 for all combinations of δ and min_{csc} except for $\text{min}_{\text{csc}} = 1$, which clearly is too lax. Second, the number of false negatives is minimized when $\text{min}_{\text{csc}} = 2$, which is the default setting in the algorithm. Third, the choice of δ

has little impact on the accuracy of the algorithm. Finally, the results for $\gamma = 15\%$ and $\gamma = 45\%$ (not shown) are highly similar to those for $\gamma = 30\%$, which means the algorithm is insensitive to the choice of γ.

We should also note two additional considerations regarding δ. First, it is sometimes necessary to use a relatively large δ, especially when the latency is high (for example, for connections that go through transcontinental or satellite links). High latency often means large variation in the delay, which can distort the keystroke timing characteristics. One possible solution to this problem would be to choose different δ's based on the RTT of a connection. This would also help with the latency-lag evasion technique discussed in § 4.4. But such adaptation complicates the algorithm, because estimating RTT based on measurements in the middle of a network path can be subtle, so we have left it for future study.

Second, large δ's also mean we must maintain state for more concurrent connection pairs, which can eat up memory and CPU cycles. Similarly, having a smaller T_{idle} means that we need to update state for connections more frequently, which in turn increases CPU consumption. To illustrate these effects, we increased δ from 80 msec to 200 msec and reduced T_{idle} from 0.5 sec to 0.3 sec. After this change, the time required to process `lbnl-telnet.trace` increases to 155 sec, more than double the 69 sec required with the current settings.

FP/FN (γ=30%)

δ (msec)	$\min_{\text{csc}}$ 1	2	4	8	12	16
20	162/0	5/0	0/0	0/2	0/5	0/6
40	683/0	19/0	0/0	0/2	0/4	0/6
80	2,486/0	134/0	0/0	0/1	0/3	0/5
120	5,633/0	431/0	12/0	3/1	3/3	2/5
160	10,131/0	995/0	28/0	7/1	4/3	2/5

Table 3: Number of false positives (FP) and false negatives (FN) for detecting indirect stepping stones when $\gamma = 30\%$.

Table 3 summarizes the results for detecting indirect stepping stones when $\gamma = 30\%$. From the table it is evident that both the number of false positives and the number of false negatives are minimized when $\min_{\text{csc}} = 4$ and $\delta \leq 80$ msec. A smaller $\min_{\text{csc}}$ or a larger δ can significantly increase the number of false positives, while a larger $\min_{\text{csc}}$ can lead to more false negatives. When $\gamma = 45\%$, the number of false positives is in general smaller, but the optimal combination of $\min_{\text{csc}}$ and δ remains the same. When $\gamma = 15\%$, the number of false positives increases 150–300%, and for $\delta = 80$ msec and $\min_{\text{csc}} = 4$, increases from 0 false positives to 7.

These findings show that the current settings of the parameters are fairly optimal, at least for the `ucb-telnet.trace`, and that there is considerable room for varying the parameters in response to certain evasion threats (§ 4.4). We also note that there is no particular need to use the same values of T_{idle}, δ, and γ for both direct and indirect stepping stones, other than simplicity, and there may be room for some further performance improvement by allowing them to be specific to the type of stepping stone, just as for $\min_{\text{csc}}$ and γ'.

5.7 Failures

In this section we summarize the common scenarios that can cause the timing-based algorithm to fail. Some of these failures have already been solved in the current algorithm, but it is beneficial to discuss them, because they illustrate some of the subtleties involved in stepping stone detection.

- Excessively short stepping stones. In many cases, the timing-based algorithm missed a stepping stone simply because the connections were exceedingly short. In some cases, the "display" and "login tag" schemes are still able to catch these because both of them key off of text sent very early during a login session.

  On the other hand, often attackers can't do very much during such short stepping stones, so failing to detect them is not quite as serious as failing to detect longer-lived stepping stones.

- Message broadcast applications such as the Unix *talk* and *wall* utilities. Such utilities can cause correlations between flows because they cause the same text to be transmitted on multiple connections. However, these correlations will be of the form $h_1 \rightarrow h_2$, $h_1 \rightarrow h_3$; that is, the connection endpoint that breaks the idle period will be the same for both flows (h_1, in this case), whereas for a true stepping stone $h_1 \rightarrow h_2 \rightarrow h_3$ the endpoint breaking the idle period will differ (first h_1, then h_2). This observation led to the directionality criterion in § 3.7.

- Correlations due to phase drift in periodic traffic. Consider two connections C_1 and C_2 that transmit data with periodicities P_1 and P_2. If the periodicities are exactly the same, then the ON/OFF periods

of the connections will remain exactly the same distance apart (equal to the phase offset for the periodicities). If, however, P_1 is slightly different from P_2, then the offset between the ON/OFF periods of the two will drift in phase, and occasionally the two will overlap. Such overlaps appear to be correlations, but actually are due to the periods being in fact *uncorrelated*, and hence able to drift with respect to one another.

This phenomenon is not idle speculation (see also [FJ94] for discussion of how it can lead to self-synchronization in coupled systems). For example, one of our traces includes two remote Telnet sessions to the same machine at the same time (involving different user IDs, but clearly the same user). The sessions had a period of overlap during which both sessions were running *pine* to check mail. For some reason, the *pine* display began periodically sending data in small chunks, with about a second between each chunk. These transmissions were initially out of sync, but sometimes sync'd up fairly closely. Before we added the rule on consecutive coincidences (parameters $\mathrm{min}_{\mathrm{csc}}$ and γ', discussed in § 4.3), these sessions had been reported as a stepping stone, because the ratio of coincidences was high enough. After we refined the algorithm, such spurious stepping stones went away (the rule on directionality discussed in the previous item would have also happened to succeed in eliminating this particular case).

- Large latency and its variation. As mentioned above, when a connection has a very high latency or large delay variation, we need to increase the value of δ (and, accordingly, γ, $\mathrm{min}_{\mathrm{csc}}$, and γ') in order to detect it. We have not yet modified the algorithm to do so because of complications in efficiently estimating a connection's RTT.

5.8 Experience with operational use

We initially expected that detecting a stepping stone would mean that with high probability we had found an intruder attempting to disguise their location. As the figures above on the frequency of detecting stepping stones indicate, this expectation was woefully optimistic. In fact, we find that wide-area Internet traffic abounds with stepping stones, virtually all of them legitimate.

For example, UCB's wide area traffic includes more than 100 stepping stones each day. These fall into a number of categories. Some are external users who wish to access particular machines that apparently trust internal UCB hosts but do not trust arbitrary external hosts. Some appear to reflect habitual patterns of use, such as "to get to a new host, type *rlogin* to the current host," in which it is not infrequent to observe a stepping stone using a remote host to access a nearby local host, or even the same local host.[2] Some are simply bizarre, such as one user who regularly logs in from UCB to a site in Asia and then returns from the Asian site back to UCB, incurring hundreds of msecs of latency (and thwarting our default choice of δ, per the above discussion). Other possible legitimate uses that we haven't happened to specifically identify are gaining anonymity for purposes other than attacks, or running particular client software provided by the intermediary but not by the upstream host.

Clearly, operational use will require development of refined Bro policy scripts to codify patterns corresponding to legitimate stepping stones, allowing the monitor to then alert only on those stepping stones at odds with the policies. But even given these hurdles, we find the utility of the algorithm clear and compelling.

Finally, we note that the detection capability has already yielded an unanticipated security bonus. Since the timing algorithm is indifferent to connection contents, it can readily detect stepping stones in which the upstream connection is made using a clear-text protocol such as Telnet or Rlogin, but the downstream connection uses a secure, encrypted protocol such as SSH. Whenever we detect such stepping stones, it is highly probable that the user typed their SSH passphrase or password in the clear over the first connection in the chain, thus undermining the security of the SSH connection. Indeed, after beginning to run the timing algorithm to look for this pattern, we rapidly found instances of such use, and confirmed that for each the passphrase was indeed typed in the clear. At LBNL, running the timing algorithm looking for such exposures is now part of the operational security policy, and, unfortunately, it continues to alert numerous times each day (and we have traced at least one break-in to a passphrase exposed in this manner at another site). Efforts are being made to educate the users about the nature of this risk.

6 Concluding remarks

Internet attackers often mask their identity by launching attacks not from their own computer, but from an inter-

[2] Inspection of some of these connections confirms that these are not inside attackers attempting to hide their location.

mediary host that they previously compromised, i.e., a stepping stone. By leveraging the distinct properties of interactive network traffic (smaller packet sizes, longer idle periods than machine-generated traffic), we have devised a stepping-stone detection algorithm based on correlating the timing of the ON/OFF periods of different connections. The algorithm runs on a site's Internet access link. It proves highly accurate, and has the major advantage of ignoring the data contents of the connections, which means both that it works for encrypted traffic such as SSH, and that the packet capture load is greatly diminished since the packet filter need only record packet headers.

While the algorithm works very well, a major stumbling block we failed to anticipate is the large number of legitimate stepping stones that users routinely traverse for a variety of reasons. One large site (the University of California at Berkeley) has more than 100 such stepping stones each day. Accordingly, the next step for our work is to undertake operating the algorithm as part of a site's production security monitoring, which we anticipate will require refined security policies addressing the many legitimate stepping stones. But even given these hurdles, we find the utility of the algorithm clear and compelling.

Finally, a natural extension to this work is to attempt to likewise detect non-interactive stepping stones, such as *relays*, in which traffic such as Internet Relay Chat [OR93] is looped through a site, and *slaves*, in which incoming traffic triggers outgoing traffic (which is not relayed), such as used by some forms of distributed denial-of-service tools [CE99]. These forms of stepping stones have different coincidence patterns than the interactive ones addressed by our algorithm, but a preliminary assessment indicates they may be amenable to detection on the basis of observing a local host that has long been idle suddenly becoming active outbound, just after it has accepted an inbound connection.

7 Acknowledgments

We would like to thank Stuart Staniford-Chen and Felix Wu for thought-provoking discussions, and in particular for the notion of deliberately introducing delay (§ 4.4); Ken Lindahl and Cliff Frost for their greatly appreciated help with gaining research access to UCB's traffic; and Mark Handley, Tara Whalen, and the anonymous reviewers for their feedback on the work and its presentation.

References

[Al94] S. Alexander, "Telnet Environment Option," RFC 1572, DDN Network Information Center, Jan. 1994.

[Bo90] D. Borman, "Telnet Linemode Option," RFC 1184, Network Information Center, SRI International, Menlo Park, CA, Oct. 1990.

[CE99] Computer Emergency Response Team, "Denial-of-Service Tools," CERT Advisory CA-99-17, Dec. 1999.

[DJCME92] P. Danzig, S. Jamin, R. Cáceres, D. Mitzel, and D. Estrin, "An Empirical Workload Model for Driving Wide-area TCP/IP Network Simulations," *Internetworking: Research and Experience*, 3(1), pp. 1-26, 1992.

[FJ94] S. Floyd, and V. Jacobson, "The Synchronization of Periodic Routing Messages," *IEEE/ACM Transactions on Networking*, 2(2), p. 122–136, April 1994.

[OR93] J. Oikarinen and D. Reed, "Internet Relay Chat Protocol," RFC 1459, Network Information Center, DDN Network Information Center, May 1993.

[PF95] V. Paxson and S. Floyd, "Wide-Area Traffic: The Failure of Poisson Modeling," *IEEE/ACM Transactions on Networking*, 3(3), pp. 226-244, June 1995.

[Pa98] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Proc. USENIX Security Symposium*, Jan. 1998.

[PR83b] J. Postel and J. Reynolds, "Telnet Option Specifications," RFC 855, Network Information Center, SRI International, Menlo Park, CA, May 1983.

[PN98] T. Ptacek and T. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Inc., http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps, Jan. 1998.

[SH95] S. Staniford-Chen and L.T. Heberlein, "Holding Intruders Accountable on the Internet." Proc. IEEE Symposium on Security and Privacy, Oakland, CA, May 1995, pp. 39–49.

[YKSRL99] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen, "SSH Transport Layer Protocol," Internet Draft, draft-ietf-secsh-transport-06.txt, June 1999.

Automated Response Using System-Call Delays

Anil Somayaji
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
soma@cs.unm.edu

Stephanie Forrest
Santa Fe Institute
Santa Fe, NM 87501
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
steph@santafe.edu, forrest@cs.unm.edu

Abstract

Automated intrusion response is an important unsolved problem in computer security. A system called pH (for process homeostasis) is described which can successfully detect and stop intrusions before the target system is compromised. In its current form, pH monitors every executing process on a computer at the system-call level, and responds to anomalies by either delaying or aborting system calls. The paper presents the rationale for pH, its design and implementation, and a set of initial experimental results.

1 Introduction

This paper addresses a largely ignored aspect of computer security—the automated response problem. Previously, computer security research has focused almost entirely on prevention (e.g., cryptography, firewalls and protocol design) and detection (e.g., virus and intrusion detection). Response has been an afterthought, generally restricted to increased logging and administrator email. Commercial intrusion detection systems (IDSs) are capable of terminating connections, killing processes, and even blocking messages from entire networks [3, 12, 22]; in practice, though, these mechanisms cannot be widely deployed because the risk of an inappropriate response (e.g., removing a legitimate user's computer from the network) is too high. Thus, IDSs become burdens, requiring administrators to analyze and respond to almost every detected anomaly. In an era of expanding connectivity and ubiquitous computing, we must seek solutions that reduce the system administrator's workload, rather than increasing it. That is, our computers must respond to attacks autonomously.

In earlier work, we and others have demonstrated several methods of anomaly detection by which large classes of intrusions can be detected, e.g., [1, 27, 17, 16]. Good anomaly detection, however, comes at the price of persistent false positives. Although more sophisticated methods will no doubt continue to be developed, we believe that it is infeasible to eliminate false positives completely. There are several reasons for this. First, computers live in rich dynamic environments, where inevitably there are new patterns of legitimate activity not previously seen by the system — a phenomenon known as *perpetual novelty* (see Hofmeyr [21] for an empirical model of the rate at which new patterns appear in a local area network). Second, profiles of legitimate activity change continually, as computers and users are added or deleted, new software packages or patches are added to a system, and so forth. Thus, the normal state of the system is *evolving* over time. Finally, there is inherent ambiguity in the distinction between normal and intrusive (or abnormal) activities. For example, changes to system configuration files are legitimate if performed by a system administrator; however, the very same actions are a security violation conducted done by a non-privileged user or an outside attacker. Thus, any automated response system must be designed to account for persistent false-positives, evolving definitions of normal, and ambiguity about what constitutes an anomaly.

We have chosen to focus on automated response mechanisms which will allow a computer to preserve its own integrity (i.e. stay "alive" and uncompromised), rather than ones that help discover the source or method of an intrusion. Within this context, we believe that the best way to approach the automated response problem is by designing a system in which a computer autonomously monitors its own activities, routinely making small corrections to maintain itself in a "normal" state. In biology, the maintenance of a stable (normal) internal environment is known as *homeostasis*. All living systems

employ a wide range of homeostatic mechanisms in order to survive under fluctuating environmental conditions. We propose that computer systems should similarly have mechanisms which strive to maintain a stable environment inside the computer, even in the face of wide variations in inputs. Under this view, automated response is recast from a monolithic all-or-nothing action (which if incorrect can have dire consequences) to a set of small, continually occurring changes to the state of the system. With this view, occasional false alarms are not problematic, because they have small impact. In earlier papers, we have advocated a view of computer security based on ideas from immunology [16, 34, 20]. This paper naturally extends that view by recognizing that immune systems are more properly thought of as homeostatic mechanisms than pure defense mechanisms [26].

In the following sections, we describe a working implementation of these ideas—a set of extensions to a Linux kernel which does not interfere with normal operation but can successfully stop attacks as they occur. We call the system pH (short for process homeostasis). To create pH, we extended our earlier intrusion-detection work using system calls [16] by connecting system calls with feedback mechanisms that either delay or abort anomalous system calls.

Delays form a natural basis for interfering with program behavior: small delays are typically imperceptible to a program, and are minor annoyances to a user. Longer delays, however, can trigger timeouts at the application and network levels, effectively terminating the delayed program. By implementing the delays as an increasing function of the number of recent anomalous sequences, pH can smoothly transition between normal execution and program termination.

This paper makes two principal contributions. First, it demonstrates the feasibility of monitoring every active process at the system-call level in real-time, with minimal impact on overall performance. Second, it introduces a practical, relatively non-intrusive method for automatically responding to anomalous program behavior.

The paper proceeds as follows. First, we review our system call monitoring and anomaly detection method. Next, we explain the design and implementation of pH. We then demonstrate pH's effectiveness at stopping attacks, show through benchmarks that it runs with low overhead, and describe what it is like to actually use pH on a workstation. After a review of related work, we conclude with a discussion of limitations and future work.

2 Background

Both the monitoring and the response components of pH use ideas introduced in [16]. What follows is a description of our original testing methodology, with which we gathered on-line data for off-line analysis. Subsequent sections explain how these techniques were modified to create pH.

To review, we monitored all the system calls (without arguments) made by an executing program on a per-process basis. That is, each time a process was invoked, we began a new trace, logging all the system calls for that process. Thus, for every process the trace consists of an ordered list (a time-series) of the system calls it made during its execution. For commonly executed programs, especially those that run with privilege, we collected such traces over many invocations of the program, when it was behaving normally. We then used the collection of all such traces (for one program) to develop an empirical model of its normal behavior.

Once the system had been trained on a sufficient number of normal program executions, the model was tested on subsequent invocations of the program. The hope was that the model would recognize most normal behavior as "normal" and most attacks as "abnormal." Our method thus falls into the category of anomaly intrusion detection.

Given a collection of system call traces, how do we use them to construct a model? This is an active area of research in the field of machine learning, and there are literally hundreds of good methods available to choose from, including hidden Markov models, decision trees, neural networks, and a variety of methods based on deterministic finite automata (DFAs). We chose the simplest method we could think of within the following constraints. First, the method must be suitable for on-line training and testing. That is, we must be able to construct the model "on the fly" in one pass over the data, and both training and testing must be efficient enough to be performed in real-time. Next, the method must be suitable for large alphabet sizes. Our alphabet consists of all the different system calls—typically about 200 for UNIX systems. Finally, the method must create models that are sensitive to common forms of intrusion. Traces of intrusions are often 99% the same as normal traces, with very small, temporally clumped deviations from normal behavior. In the following, we describe a simple method, which we call "time-delay embedding" [16]. Warrender [38] compared time-delay embedding with several other common machine learning algorithms

and discovered that it is remarkably accurate and efficient in this domain.

We define normal behavior in terms of short n-grams of system calls. Conceptually, we define a small fixed size window and "slide" it over each trace, recording which calls precede the current call within the sliding window. The current call and a call at a fixed preceding window position form a "pair," with the contents of a window of length x being represented by $x-1$ pairs. The collection of unique pairs over all the traces for a single program constitutes our model of normal behavior for the program.[1]

More formally, let

$$
\begin{array}{rcl}
S & = & \text{alphabet of possible system calls} \\
T & = & \text{trace} \\
 & = & \text{the sequence } t_0, t_1, \ldots, t_{\tau-1}, t_i \in S \\
w & = & \text{window size}, 2 \leq w \leq \tau \\
P & = & \text{profile} \\
 & = & \text{set of patterns associated with } T \text{ and } w \\
 & = & \{\langle s_i, s_j \rangle_k : s_i, s_j \in S, 1 \leq k < w \\
 & & \qquad \exists p : \; 0 \leq p < \tau - k, \\
 & & \qquad\quad t_p = s_i, \\
 & & \qquad\quad t_{p+k} = s_j,
\end{array}
$$

For example, suppose we had as normal the following sequence of calls:

execve, brk, open, fstat, mmap, close, open, mmap, munmap

and a window size of 4. We slide the window across the sequence, and for each call we encounter, we record what call precedes it at different positions within the window, numbering them from 0 to $w-1$, with 0 being the current system call. So, for this trace, we get the following windows:

position 3	position 2	position 1	current
			execve
		execve	brk
	execve	brk	open
execve	brk	open	fstat
brk	open	fstat	mmap
open	fstat	mmap	close
fstat	mmap	close	open
mmap	close	open	mmap
close	open	mmap	munmap

When a call occurs more than once in a trace, it will likely be preceded by different calls in different contexts. We compress the explicit window representation by joining together lines with the same current value (note the open and mmap rows):

current	position 1	position 2	position 3
execve			
brk	execve		
open	brk, close	execve, mmap	fstat
fstat	open	brk	execve
mmap	fstat, open	open, close	brk, mmap
close	mmap	fstat	open
munmap	mmap	open	close

This table can be stored using a fixed-size bit array. If $|S|$ is the size of the alphabet (number of different possible system calls) and w is the window size, then we can store the complete model in a bit array of size: $|S| \times |S| \times (w-1)$. Because w is small (6 is our standard default), our current implementation uses a 200×200 byte array, with masks to access the individual bits.

At testing time, system call pairs from test traces are compared against those in the normal profile. Any system call pair (the current call and a preceding call within the current window) not present in the normal profile is called a *mismatch*. Any individual mismatch could indicate anomalous behavior (a true positive), or it could be a sequence that was not included in the normal training data (a false positive). The current system call is defined as anomalous if there are any mismatches within its window.

To date, all of the intrusions we have studied produce anomalous sequences in temporally local clusters. To facilitate the detection of these clusters, we record recent anomalous system calls in a fixed-size circular array, which we refer to as a *locality frame*. More precisely, let n be the size of our locality frame, and let A_i be the i-th entry of the locality frame array, with $0 \leq i < n$ and $A_i \in \{0, 1\}$. Then, for system call s ($0 \leq s < \tau$) with mismatches m_s, $A_{s \bmod n} = 1$ iff $m_s > 0$, and is 0 other-

[1] Our original paper on using system calls for intrusion detection [16] used a technique called "lookahead pairs." pH uses the original lookahead pairs algorithm as described here, except that it looks behind instead of ahead. Later papers [20, 38] report results based on recording full sequences. We reverted to lookahead pairs because it is simple to implement and extremely efficient.

wise. Thus, the locality frame implicitly stores the number of the past n system calls which were anomalous. We call this total of recent anomalies, $\sum A_i$, the locality frame count (LFC).[2] For the experiments described below, we used a locality frame of size 128.

3 pH Design

pH performs two important functions: It monitors individual processes at the system-call level, and it automatically responds to anomalous behavior by either slowing down or aborting system calls. Normal behavior is determined by the currently running binary program; response, however, is determined on a per-process basis.

To minimize I/O requirements and maximize efficiency, stability, and security, we have implemented most of pH in kernel space. We considered several alternative approaches, including audit packages, system-call tracing utilities (such as `strace`), and instrumented libraries. However, each of these other approaches has serious drawbacks. Audit packages generate voluminous logfiles, which are expensive to create and even more expensive to analyze. Additionally, they do not routinely record every system call. User-space tracing utilities are too slow for our application, and in some cases, they interfere with privileged daemons to the extent that they behave incorrectly. Instrumented libraries cannot detect every system call, because not every system call comes through a library function (e.g., buffer overflow attacks). In addition, a kernel implementation allows us to put our monitoring and response mechanisms exactly where they are needed, in the system call dispatcher, and allows the implementation to be as secure as the kernel.

For each running executable, pH maintains two arrays of pair data: A training array and a testing array. The training array is continuously updated with new pairs as they appear; the testing array is used to detect anomalies, and is never modified except by replacing it with a copy of the training array. Put another way, the testing array is the current normal profile for a program, while the training array is a candidate future normal profile.

A new "normal" is installed by replacing the testing array with the current state of the training array. The replacement occurs under three conditions: (1) the user explicitly signals via a special system call (`sys_pH`) that a profile's training data is valid; (2) the profile anomaly count exceeds the parameter $anomaly\_limit$; (3) the training formula is satisfied. When an anomaly is detected, the current system call is delayed according to a simple formula. Details of these conditions and actions are given in the next several paragraphs.

The training to testing copy can occur automatically based on the state of the following training statistics:

$$\begin{aligned} train\_count: &\quad \text{\# calls since array initialization} \\ last\_mod\_count: &\quad \text{\# calls since array was last modified} \\ normal\_count = &\quad train\_count - last\_mod\_count \end{aligned}$$

When the training array meets all of the following conditions, it is copied onto the testing array (note: this is the normal mechanism for initiating anomaly detection in the system):

$$\begin{aligned} last\_mod\_count &> mod\_minimum \\ normal\_count &> normal\_minimum \\ \frac{train\_count}{normal\_count} &> normal\_ratio \end{aligned}$$

The three parameters on the right are user defined, and can be set at runtime.

As we mentioned earlier, pH responds to anomalies by delaying system call execution. The amount of delay is an exponential function of the current LFC, regardless of whether the current call is anomalous or not. The unscaled delay for a system call is $d = 2^{LFC}$. The effective delay for a system call is $d \times delay\_factor$, where $delay\_factor$ is another user-defined parameter. Note that delays may be disabled by setting $delay\_factor$ to 0. If the LFC ever exceeds the $tolerization\_limit$ parameter (which is 12 for the experiments described below), the training array is reset, preventing truly anomalous behavior from being incorporated into the testing array.

Because pH monitors process behavior based on the executable that is currently running, the execve system call causes a new profile to be loaded. Thus, if an attacker were able to subvert a process and cause it to make an execve call, pH might be tricked into treating the current process as normal, based on the data for the newly-loaded executable. To avoid this possibility the maximum LFC count (maxLFC) for a process is recorded. If

[2] A somewhat different approach was taken in Hofmeyr [20], where the measure of anomalous behavior was based on Hamming distances between unknown sequences and their closest match in the normal database.

maxLFC exceeds the *abort_execve* threshold, then all execve's are aborted for the anomalous process.

pH also keeps a count of the raw number of anomalies each profile has seen. This count can be seen as a measure of ongoing, non-clustered abnormal behavior. If this number exceeds the parameter *anomaly_limit*, pH automatically copies the training array to the testing array, causing pH to treat similar future behavior as normal. Borrowing from immunology, we refer to this process as *tolerization*. Low values of *anomaly_limit* allow pH to automatically tolerize most novel behavior, while higher values inhibit tolerization. When a system is initially set up, automatically-created normal profiles may contain too little normal behavior. To reduce the number of reported anomalies, *anomaly_limit* should be set to a small value (less than 10). Then, once the system has stabilized, *anomaly_limit* should be set to at least 20 to prevent pH from automatically learning the behavior of attacks.

4 Implementation

The pH prototype is implemented as a patch for the Linux 2.2 kernel, and was developed and tested on systems running a pre-release of the Debian/GNU Linux 2.2 distribution [35]. The modified kernel is capable of monitoring every executed system call, recording profiles for every executable. An overview of the system is shown in Figure 1.

Program profiles for each executable are stored on disk. Each profile contains both a training and testing array, and so is actually two "profiles" by the terminology in Section 2. The kernel loads the current profile when a new program begins executing (on execve), and then writes it out again when the process terminates. When a new executable is loaded via the execve system call, the kernel attempts to load the appropriate profile from disk; if it is not present, a new profile is created. If another process runs the same executable, the profile is shared between both processes. To prevent consistency problems due to interleaving, each executing process maintains its own record of recent system calls (its current sequence). When all processes using a given profile terminate, the updated profile is saved to disk. A loaded profile consumes approximately 80K of kernel (non-swappable) memory.

We modified the system call dispatcher so that it calls a pH function (`pH_process_syscall`) prior to dispatching the system call. `pH_process_syscall` implements the monitoring, response, and training logic. pH is controlled through its own system call, `sys_pH`, which allows the superuser (root) to take the following actions:

- Start, stop monitoring processes.
- Set system parameters (see Section 3 for descriptions):
  - *delay_factor*
  - *abort_execve*
  - *mod_minimum*
  - *normal_minimum*
  - *normal_ratio*
  - *tolerization_limit*
  - *anomaly_limit*
- Turn on/off logging of system calls to disk (expensive, used for debugging).
- Turn on/off logging novel sequences to disk.
- Status (prints out current values of system parameters to the kernel log).
- Write all profiles to disk.
- Reset <pid>: Resets the profile to be empty.
- Start normal <pid>: Copies the training array for *pid*'s executable to its testing array, and marks the profile as normal.
- Tolerize <pid>: Change the normal flag for *pid*'s profile to 0, reset its locality frame, and cancel any current delay for it.
- Sensitize <pid>: Clears the training array. This mechanism is used to prevent known true positives from being incorporated into the training data.
- Turn on/off debugging messages sent to kernel logging facility.

More specifically, we extended the Linux task structure (the kernel data structure used to represent processes and kernel-level threads) with a new structure which contains the following fields: the current window of system calls for the task, a locality frame, and a pointer to the current profile. A profile is a structure containing two byte-arrays for storing pairs (the training and testing arrays) and some additional training statistics described in Section 3.

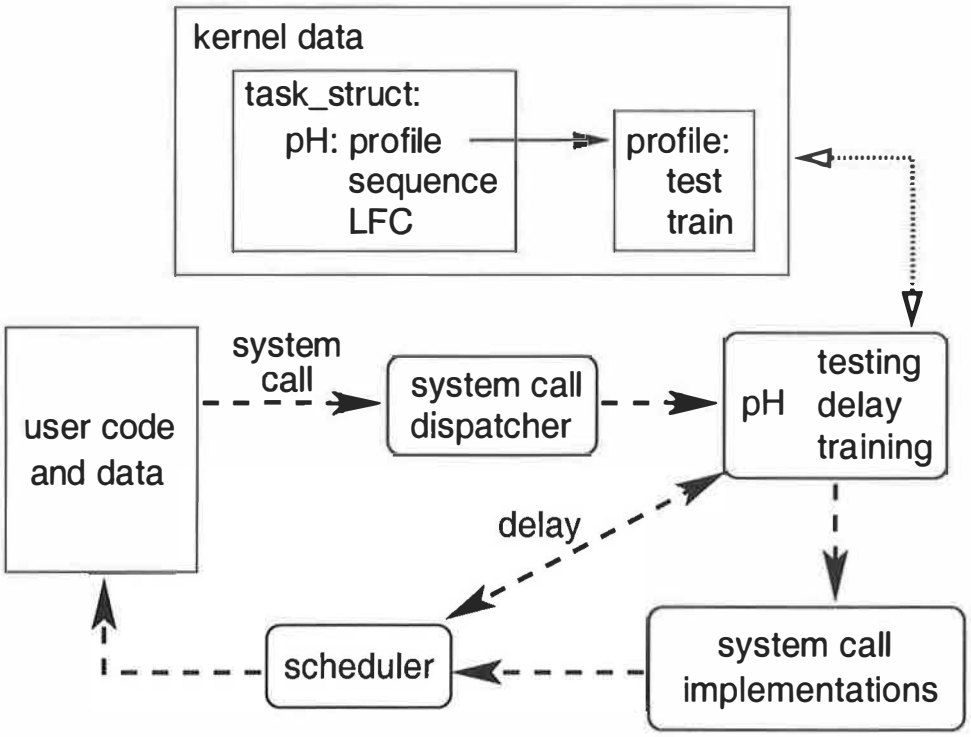


Figure 1: Basic flow of control and data in a pH-modified Linux kernel.

5 Experimental Results

In this section, we report on some early experiments testing out pH in a live environment. We are interested in three aspects of the system: Its effectiveness in intrusion response (can it really detect and stop an attack before the system is compromised?), performance impact (what is the overhead of the installed system?), and usability (what is it like to live with pH on your own computer?).

5.1 Can pH detect and stop attacks in time to prevent system compromise?

To test how pH could respond to security violations, we tested its behavior by seeing how it could detect and respond to a Secure Shell (SSH) daemon [29] backdoor, an SSH daemon buffer overflow, and a sendmail [13] attack that exploits a bug in the Linux kernel's capabilities code. These three violations all allow an attacker to obtain root privileges, using different techniques to gain access. Delays alone are significant inhibitors of these attacks; with execve aborts, pH can effectively stop all of them.

To test the SSH attacks, the `sshd` program in Debian 2.2's packaged version of Secure Shell (ssh-nonfree), version 1.2.27-6 was modified in two basic ways. First, it was made to link against the RSAREF2 library, to make it vulnerable to a buffer overflow attack script published on the BUGTRAQ mailing list [2]. Second, the source was modified using the rkssh5 trojan patch [37], and was built using the "global password" flag. This option allows an attacker to access to any account on the system using a compiled in, MD5-encoded password. In addition, use of this password disables most logging, minimizing the evidence of the intrusion.

A normal profile for this modified `sshd` binary was created by exercising the program on a personal workstation. Normal logins via root and a regular user were tested, using the password, RSA-secured rhosts, and pure RSA methods of authentication. Also, failed logins were tested, using nonexistent users and incorrect passwords. Together these produced 687 sequences, and a profile with 1725 pairs, over 47756 system calls.

Relative to this synthetic normal profile, we first tested whether pH could detect the use of the global password to gain access to the root account. With all responses disabled, the backdoor produced 5 anomalies, 3 in the child (which exec's the remote user's shell), and 2 in the

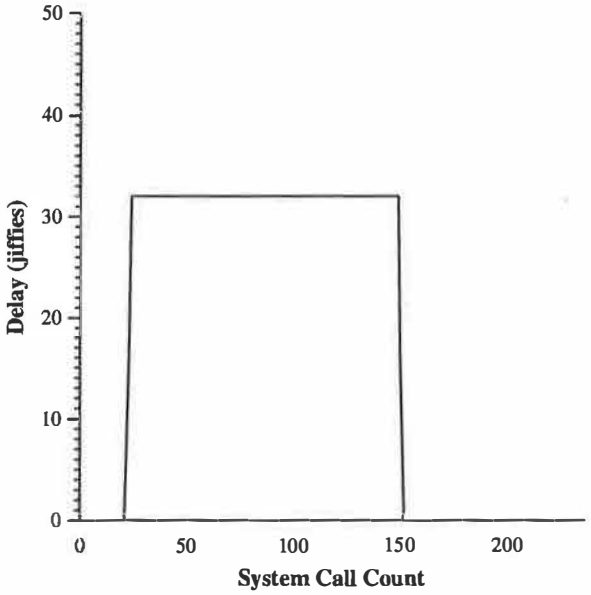


Figure 2: A graph showing the pH-induced system-call delay during the sshd backdoor intrusion. Note the exponential increase (from 0 to 8, 16, then 32) and decrease, with a constant delay for most calls within the locality frame. The process shown is the child process, and it terminates with a shell being exec'd. The pair window size is 6, the locality frame size is 128, and the *delay_factor* is set to 4. Time is measured in jiffies, which are 1/100 of a second on Linux running on i386-compatible machines.

parent (which maintains the network connection). Setting *delay_factor* to 4 produced the same anomaly profile, but did not prevent the remote user from logging in; however, the resulting connection was slowed down significantly, as shown in Figure 2. With *abort_execve* set to 1, the backdoor was closed, whether delays were enabled or not.

With all responses disabled, the buffer overflow attacked produced 4 clustered anomalies in the parent SSH process. Setting *delay_factor* to 4 produced the same anomalies, and allowed the attacker to obtain a root shell; however, this shell was less useful than might be supposed. Recall that pH delays every process with a non-zero LFC, and the LFC is only updated if the program has a valid normal (test) profile. As it turns out, bash, the standard shell on most Linux systems, is a large, complicated program that tends not to reach a stable profile. Thus, the 64 jiffy (0.64s) delay incurred by the overflowed sshd is passed on to the exec'd bash, and bash keeps this delay forever! Even if this weren't the case, because of the 128 entry locality frame, we'd see the delay for 125 system calls, giving us at least an 80s delay. Not a huge amount of time, but possibly enough to make a cracker think the attack isn't working.

With execve aborts enabled, the overflow attack was stopped, whether delays were enabled or not. The attack script does not simply fail, though; since the overflow code keeps retrying the execve call if it fails, the aborts cause an infinite loop. Each pass through the loop generates 3 anomalies, due to the failed execve; a few times through this tight loop thus causes the *tolerization_limit* to be exceeded, causing sshd's training profile to be reset.

The Linux capability vulnerability allows a non-privileged program to prevent a privileged one from dropping its capabilities on systems running recent 2.2 kernels (2.2.14 and 2.2.15 are both vulnerable). An exploit was published on BUGTRAQ [28] which uses sendmail to take advantage of this hole. Because this is a flaw in the kernel, it can succeed even though sendmail does the right thing and tries to drop its privileges.

A normal profile for sendmail (Debian version 8.9.3-22) was first generated, based on normal usage on a personal workstation. This normal had 3443734 system calls with 1061 unique sequences, and produced a profile with 2412 system call pairs. Relative to this normal, the exploit was extremely noticeable. The exploit generates several different sendmail processes, and just one of them had 47 anomalies! Indeed, the numerous anomalies caused the *tolerization_limit* to be reached

numerous times. Enabling execve aborts did nothing to inhibit the attack; this makes sense, since the exploit doesn't have sendmail directly run a privileged shell; instead, it creates a setuid-root shell in /tmp. However, a *delay_factor* of 4 effectively stopped the attack — delays were produced which lasted for at least two hours. Time delays of this magnitude would almost certainly frustrate a normal cracker; a patient one could be addressed by automatically killing any process that had been delayed for a long time period, say 30 minutes or more.

5.2 What is the overhead of running pH?

To determine the performance impact of our kernel modifications, we ran the HBench-OS 1.0 [11] low-level benchmark suite on an HP Pavilion 8260 (266 MHz Pentium II, 160M SDRAM, Maxtor 91020 10G Ultra-DMA IDE hard disk) running a pre-release version of Debian/GNU Linux 2.2. Tests were run for ten iterations on a system running in single user mode. In Tables 1 and 2, "Standard" refers to a stock Linux 2.2.14 kernel. "pH" refers to a 2.2.14 kernel with pH extensions, with monitoring enabled for all processes and with status messages and automated response turned off. All times are in microseconds.

Tables 1 and 2 show that our modifications add significantly to system call overhead. Table 1 indicates that pH adds approximately 4.7 μs to the execution time of simple system calls that normally would take less that 2 μs to execute. Table 2 shows that pH causes process creation to be almost twice as slow for a dynamically-linked shell. Although these tables show a significant performance hit, they are not indicative of the impact on overall system performance.

Table 3 shows how overall performance is affected for a set of tasks. Here we report the output of `time` for three different kinds of operations: kernel builds, `find / -print > /dev/null` (a basic traversal of the file system), and Quake 2 frame rates. All of these tests were run in single-user mode. The most dramatic effect is seen in the system time of the kernel build, which almost doubles due to monitoring overhead. This difference, however, only causes a 4% slowdown in the clock time. The find test shows almost a 10% slowdown, and this is for a program that is almost entirely bound by the speed of filesystem-access system calls. Interestingly, the Quake 2 frame rate tests shows virtually no slowdown. These tests illustrate what we have observed informally by using the system ourselves: If delays are

System Call	Standard (μs)	pH (μs)
getpid	1.1577 (0.00000)	5.8898 (0.00025)
getrusage	1.9145 (0.00000)	6.6137 (0.00138)
gettimeofday	1.6703 (0.00184)	6.3779 (0.00112)
sigaction	2.5609 (0.00010)	7.2928 (0.01029)
write	1.4135 (0.00187)	6.1637 (0.00075)

Table 1: System call latency results. All times are in microseconds. Standard deviations are listed in parentheses.

Operation	Standard (μs)	pH (μs)
null	408.80 (00.618)	2497.90 (40.923)
simple	2396.24 (11.124)	8206.62 (11.795)
/bin/sh	9385.66 (26.761)	18223.96 (26.777)

Table 2: Dynamic process creation latency results. Null refers to a fork of the current process. Simple is a fork of the current process plus an exec() of a hello-world program written in C. /bin/sh refers to the execution of hello-world through the libc system() interface, which uses /bin/sh to invoke hello-world. All times are in microseconds. Standard deviations are listed in parentheses.

turned off, a user can use the modified workstation without noticing any differences in system behavior, even if she decides to run a compute and I/O intensive application such as Quake 2.

5.3 pH in Practice

To understand the usability of the prototype, the modified kernel was installed on the authors' personal computers, configured to monitor every process on the system. As indicated above, such a configuration has a minimal performance impact in practice; however, enabling delays in this situation can cause certain problems. Privileged programs, such as `login`, `sendmail`, and `cron`, have a highly constrained behavior profile; thus, after a day or two of sampling, these programs tend to settle into a stable normal, and exhibit few anomalies. Large non-privileged programs, such as netscape and emacs, have more complicated behaviors, and thus tend not to shift into a normal monitoring mode, and so are never delayed.

Some of the more interesting programs are ones which perform simple system monitoring, such as `asclock` (a NeXTStep-style clock) and `wmapm` (a battery monitoring program). These programs execute a large number of

Benchmark	Standard	pH
kernel build (s)		
real	702.47 (0.07)	727.44 (0.29)
user	669.35 (0.60)	673.67 (0.55)
sys	33.00 (0.61)	53.60 (0.70)
find / -print (s)		
real	5.68 (0.58)	6.24 (0.54)
user	1.61 (0.09)	1.59 (0.09)
sys	3.27 (0.09)	3.90 (0.17)
Quake 2 (fps)		
demo1	22.89 (0.03)	22.87 (0.05)
demo2	23.30 (0.00)	23.30 (0.00)

Table 3: Overall system performance. All units are seconds (s), except for the Quake 2 test, which is in frames per second (fps). Ten trials were run for each example, except 100 trials were run for `find`. Each test was run once before beginning the measurements in order to eliminate initial I/O transients. Standard deviations are listed in parentheses.

system calls, and most of the time they have repetitious behavior. However, when a user perturbs the system by changing desktops or by moving windows, the behaviors of these programs can change. In the current prototype, these programs tend to be the first to obtain normals, and the first to be slowed down. Over a few days they tend to settle down and operate normally; this transition, however, can require a number of user-supplied tolerization events. This suggests that the heuristics described in Section 3 may need to be refined. However, by temporarily setting $anomaly\_limit$ to a low value (such as 5), the number of reported anomalies can be kept to a minimum.

As monitoring programs are generally not critical applications, problems involving them can be seen as minor nuisances. A more significant set of issues arises with the behavior of one large, privileged program: the X server. The X server is responsible for initializing and controlling access to the video display, on behalf of X clients such as `xterm` and netscape. X servers are similar to monitoring programs, in that they make a large number of mostly-repetitive system calls, and so tend to acquire a normal profile quickly. User actions can also perturb the X server's behavior, causing it to be delayed. In this case, the delays can have dramatic effects, such as causing a user's entire session to be frozen or leaving the video hardware in a confused state when they occur during server initialization or shutdown. Fortunately, most of these problems can be avoided by initially starting up and shutting down the X server a few times, allowing pH to learn the critical initialization and shutdown system call patterns.

These two classes of problems suggest a weakness in our current approach. Programs which make large numbers of system calls in a short period of time tend to acquire normal profiles, even when a true sampling of behavior has not yet occurred. A natural solution is to take time into account during the normal profile decision process. Such a strategy might require a significant amount of computation, and so is probably better implemented in a userspace control daemon. It would also allow additional factors to be considered, such as size of executable, number of invocations, and perhaps program-specific heuristics. Such a daemon is planned for the future.

6 Related Work

Our approach to homeostatic computing is similar in spirit to Brooks' approach to mobile robot control, based on loosely coupled feedback loops, real-time interactions with dynamic environments, and no centralized representation of the outside world [9, 10]. We believe Brooks' subsumption architecture can be applied to the construction of a computer security system. pH in its current form is analogous to feedback loops that help a robot maintain balance; with the addition of a parameter-adjusting control daemon, we may be able to teach pH how to "walk."

Although research IDSs have performed anomaly detection for years [1, 27, 17, 16], most commercial systems emphasize misuse detection (i.e. pattern matching for known attacks), requiring frequent updates as new exploits are developed. Many current commercial network IDSs [3, 12, 22] are capable of automatically responding to network attacks through increased logging, firewall reconfiguration, termination of connections, and even automatic blocking of suspicious networks. Combined host and network IDSs such as ISS RealSecure [22] can also respond to threats by terminating individual processes. However, because responses that halt attacks can also cause significant service reductions, these responses must be reserved for attacks which can be easily and reliably identified through specific misuse signatures. Although useful for high-security installations, actions such as session capture and email/pager notification are simply a burden to most administrators.

Sekar, Bowen, and Segal [30] have developed a specification-based approach for intrusion detection and

automated response at the system-call level. They have created a language called ASL for specifying program behavior and responses to abnormal behavior, and they have created Linux kernel extensions which allow their specifications to be enforced on-line. Their approach has the advantage of allowing subtle responses to security violations, ranging from changing system call arguments to confining a program to an alternative file system. Unfortunately, it also has the disadvantages of being labor-intensive, in that specifications must be constructed manually for each executable.

Michael Ernst and others at the University of Washington have developed techniques for dynamically determining program invariants [15]. pH also dynamically detects invariants in program behavior, although it does so at the system-call instead of the data-structure level. Perhaps Ernst's techniques could be used to create an on-line data monitoring tool which would complement the system-call monitoring of pH.

Delays are used throughout computing to achieve varying goals. Most laptop CPUs have the ability to run at a slower speed to minimize heat or maximize battery life; Transmeta's Crusoe processor [14] goes a step further by allowing the speed of the chip to vary continuously in response to system load, maximizing battery life and perceived performance. The Ethernet protocol arbitrates wire access by having transmitting computers exponentially delay their packets when collisions are detected [36]. And, at the software level, the standard `login` program on most UNIX systems delays repeated login attempts to interfere with password guessing attacks. A final example is the program `getty`, which notices if it spawns processes too frequently on a given tty device and in this event, puts itself to sleep for a few minutes.

The core of pH can be seen as an unusual type of process scheduling. In most UNIX systems [4], processes are scheduled using static priorities (provided by the administrator), dynamic priorities (based on recent CPU and I/O behavior), and the number of processes on the system. "Fair share" schedulers divide CPU time between users, not processes [18, 24]. pH's delay mechanism could be viewed as an implicit mechanism for allocating CPU time; however, instead of being fair to all processes or users, it favors processes which are behaving "normally."

Research on high-performance operating systems emphasizes extensible [5, 31] and minimal [23] kernels. These systems require novel security mechanisms to moderate the increased power given to application programs, relative to operating systems with conventional, monolithic kernels. In contrast, our work on biologically-inspired OS extensions assumes a conventional kernel, and aims to increase the stability and security of the system.

Adaptive, on-line control has been widely studied as a method for improving system performance. Whether motivated by non-stationary workloads [7], extensible operating systems [32], parallelism [25], or on-line database transaction processing [39], researchers have focused on using adaptive methods for improving system performance, not robustness. Work in using adaptive control in real-time systems [6] has focused on using adaptation to help meet timing and robustness constraints.

Finally, pH can be seen as a type of fault tolerant system [8, 33, 19], except that we focus on security violations instead of hardware or software failures.

7 Discussion

A major point of this paper is that it is feasible to use system-call delays to stop intrusions in real-time, without prior knowledge about what form an attack might take (unlike signature-based scanners). The three example exploits help show that pH can do this, even for very different types of attacks. However, in practice pH's effectiveness is determined by whether it can obtain stable normals for the binaries on a system. Currently, pH can do this automatically only for programs which are relatively simple and are called on a regular basis; even then, there is an ongoing risk that pH could be trained to accept intrusions as normal behavior. Research still needs to be done on more effective training heuristics that minimize the time for pH to obtain a normal profile, but also minimize the chances of pH tolerizing truly abnormal behavior. By incorporating such heuristics into a pH control daemon, we should be able to minimize the need for user or administrator intervention.

It may be necessary to implement a default timeout mechanism through pH, in which any process that is delayed beyond a certain point is automatically terminated. It may also be necessary to increase pH's repertoire to include actions such as system call parameter modifications. Additional response mechanisms may require computationally expensive analysis algorithms to be added; because abnormally-behaving processes are delayed, pH actually has the time to perform more sophisticated analysis when anomalies are detected. Our

philosophy, however, is to wait until such a need arises before implementing additional mechanisms.

A second major point of the paper is to show that system-call monitoring is practical, even when every executing process on the system is monitored simultaneously. pH routinely monitors every system call executed by every process with little perceptible overhead. Thus, we believe that the current implementation of pH is efficient enough to satisfy a wide variety of users.

The current version of pH is not completely secure. pH does restrict use of the `sys_pH` system call to users who have the kill capability (which, by default is only root); however, there are no checks to ensure that a profile has not been tampered with on disk, or restrictions on user access to profiles — they are currently owned by root, but readable by anyone. An attacker could use this information to design a less-detectable attack based on the system call usage on the target machine. pH could be used to generate a denial-of-service attack by triggering abnormal (but otherwise benign) behavior in a target program. Also, it may be useful to implement mechanisms to prevent users (including root) from being able to directly modify the stored profiles. Such "hardening" of pH, though, should wait until pH's basic functionality has undergone further testing.

In the past, we have emphasized that system call profiling is a suitable technique for monitoring privileged programs. pH in its current form, however, monitors and responds to anomalies in all programs. In the future, we may decide to restrict monitoring to privileged programs; yet, with the increasing use of active content on the Internet, it may also be desirable to have pH respond to anomalies in word processors and web browsers. Some large programs such as netscape are implemented using userspace threads, causing system calls to be interleaved in apparently random patterns due to variations in thread scheduling; thus, the system call profiles of these programs may never stabilize. We believe, though, that this will be less of a problem in the future, as programs switch to using kernel threads. Because the Linux kernel uses the same data structure to represent threads and processes, pH is able to monitor kernel threads individually, avoiding interleaving effects.

8 Acknowledgments

The authors gratefully acknowledge the support of the National Science Foundation (grant IRI-9711199), the Office of Naval Research (grant N00014-99-1-0417), and the Intel Corporation.

Steven Hofmeyr wrote the original program for analyzing system call traces, Julie Rehmeyr rewrote the code so that it was suitable to run in the kernel, and Geoff Hunsicker developed the original login trojan, which we ported for these experiments. Margo Seltzer suggested some of the benchmarks used in the paper. Erin O'Neill pointed out to us that the immune system is better thought of as a system for maintaining homeostasis than as a defense mechanism. We are grateful to the above people and all the members of the Adaptive Computation group at UNM, especially David Ackley, for their many helpful suggestions and interesting conversations about this work.

9 Availability

The current version of pH may be obtained via the following web page:

http://www.cs.unm.edu/~soma/pH/

The distribution contains a kernel patch and a few support programs. All are licensed under the terms of the GNU General Public License (GPL).

References

[1] Debra Anderson, Thane Frivold, and Alfonso Valdes. Next-generation intrusion detection expert system (NIDES): A summary. Technical Report SRI–CSL–95–07, Computer Science Laboratory, SRI International, May 1995.

[2] Ivan Arce. SSH-1.2.27 & RSAREF2 exploit. BUGTRAQ Mailing list (bugtraq@securityfocus.com), December 14 1999. Message-ID: <3856C3EF.230F0AE@core-sdi.com>.

[3] Axent Technologies, Inc. Netprowler. http://www.axent.com, 2000.

[4] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[5] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, Copper Mountain, CO, 1995.

[6] Thomas E. Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[7] P.R. Blevins and C.V. Ramamoorthy. Aspects of a dynamically adaptive operating system. *IEEE Transactions on Computers*, 25(7):713–725, July 1976.

[8] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under unix. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[9] Rodney A. Brooks. A robust layered control system for a mobile robot. A.I. Memo 864, Massachusetts Institute of Technology, September 1985.

[10] Rodney A. Brooks and Anita M. Flynn. Fast, cheap, and out of control: a robot invasion of the solar system. *Journal of The British Interplanetary Society*, 42:478–485, 1989.

[11] A. Brown and M. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, June 1997.

[12] Cisco Systems, Inc. Cisco secure intrusion detection system. http://www.cisco.com/warp /public/cc/cisco/mkt/security/nranger/tech /ntran_tc.htm, 1999.

[13] Sendmail Consortium. sendmail.org. http://www.sendmail.org/, 2000.

[14] Transmeta Corporation. Crusoe processor: Longrun technology. http://www.transmeta.com /crusoe/lowpower/longrun.html, January 2000.

[15] Michael D. Ernst, Adam Czeisler, William G. Griswold, , and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 7–9 2000.

[16] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press, 1996.

[17] L. T. Heberlein, G. V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Press, 1990.

[18] G.J. Henry. The fair share scheduler. *Bell Systems Technical Journal*, 63(8):1845–1857, October 1984.

[19] M. A. Hiltunen and R. D. Schlichting. Adaptive distributed and fault-tolerant systems. *Computer Systems Science and Engineering*, 11(5):275–285, September 1996.

[20] S. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[21] Steven A. Hofmeyr. *An Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, University of New Mexico, 1999.

[22] Internet Security Systems, Inc. RealSecure 3.0. http://www.iss.net, 1999.

[23] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malô, France, October 1997.

[24] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

[25] D.M. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.

[26] Erin O'Neill. Personal Communication, October 1998.

[27] P. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings National Information Systems Security Conference*, 1997.

[28] Wojciech Purczynski. Sendmail & procmail local root exploits on Linux kernel up to 2.2.16pre5. BUGTRAQ Mailing list (bugtraq@securityfocus.com), June 9 2000. Message-ID: <Pine.LNX.4.21.0006090852340.3475-300000@alfa.elzabsoft.pl>.

[29] SSH Communications Security. SSH secure shell. http://www.ssh.com/products/ssh/, 2000.

[30] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*. The USENIX Association, April 1999.

[31] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation (OSDI II)*, 1999.

[32] Margo Seltzer and Christopher Small. Self-monitoring and self-adapting systems. In *Proceedings of the 1997 Workshop on Hot Topics on Operating Systems*, Chatham, MA, May 1997. http://www.eecs.harvard.edu/~vino/vino/papers/monitor.html.

[33] E. Shokri, H. Hecht, P. Crane, J. Dussault, and K.H. Kim. An approach for adaptive fault-tolerance in object-oriented open distributed systems. *International Journal of Software Engineering and Knowledge Engineering*, 8(3):333–346, September 1998.

[34] A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a computer immune system. In *New Security Paradigms Workshop*, New York, 1998. Association for Computing Machinery.

[35] SPI. Debian. http://www.debian.org/, 2000.

[36] Andrew S. Tanenbaum. *Computer Networks*, chapter 3, pages 145–146. Prentice Hall PTR, Englewood Cliffs, NJ, 2nd edition, 1989.

[37] timecop. Root kit SSH 5.0. http://www.ne.jp/asahi/linux/timecop/, January 2000.

[38] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, Los Alamitos, CA, 1999. IEEE Computer Society.

[39] G. Weikum, C. Hasse, A. Monkeberg, and P. Zabback. The COMFORT automatic tuning project. *Information Systems*, 19(5):381–432, July 1994.

CenterTrack: An IP Overlay Network for Tracking DoS Floods

Robert Stone
UUNET Technologies, Inc.
robert@uu.net

Abstract

Finding the source of forged Internet Protocol (IP) datagrams in a large, high-speed network is difficult due to the design of the IP protocol and the lack of sufficient capability in most high-speed, high-capacity router implementations. Typically, not enough of the routers in such a network are capable of performing the packet forwarding diagnostics required for this. As a result, tracking-down the source of a flood-type denial-of-service (DoS) attack is usually difficult or impossible in these networks.

CenterTrack is an overlay network, consisting of IP tunnels or other connections, that is used to selectively reroute interesting datagrams directly from edge routers to special tracking routers. The tracking routers, or associated sniffers, can easily determine the ingress edge router by observing from which tunnel the datagrams arrive. The datagrams can be examined, then dropped or forwarded to the appropriate egress point.

This system simplifies the work required to determine the ingress adjacency of a flood attack while bypassing any equipment which may be incapable of performing the necessary diagnostic functions.

1 Introduction

While the Internet Protocol is simple and effective, it lacks an obvious way to ensure that the address contained in the source address field of each packet is actually representative of where the packet originated [1]. IP alone does not address this security issue. Consequently, an attacker can forge the source address of an IP packet from any site on the Internet which is not subject to some sort of source validation mechanism. Examples of such mechanisms are egress filtering (by the originating site) and ingress filtering (by the backbone provider) [2]. As of the time of this writing, source address validation is not yet in general practice on the Internet.

Furthermore, there are no quality-of-service or resource restriction mechanisms that are practical or in general use that prohibit an attacker from consuming all available bandwidth on a network connection.[1] Due to their trivial nature, packet flood attacks are not only possible, but they have become quite common.

A flood attack differs from other types of DoS attacks in that it requires constant and rapid transmission of packets to the victim in order to be effective. Some DoS attacks, such as the "ping of death" [3] and similar attacks, merely require that a single "killer packet" reach the victim. This paper addresses the issue of tracking (tracing) *flows* of forged packets rather than individual forged packets. Tracking individual forged packets is generally infeasible and thus other methods, such as patching vulnerable systems, are commonly used to protect against the killer packet class of attacks.

Internet Service Providers (ISPs) and their customers are frequently the victims of various types of packet flood attacks. Included in the category of packet flood attacks are "Smurf" [4], "Fraggle" [5], TCP SYN Flood [6], and others. Not only is it possible for the attacker to arbitrarily alter the source address field of the attack packets without penalty, many of these attacks actually require that the source address be faked in order to be effective.

Because the source address field is almost always forged in these attacks, it is non-trivial to determine their true origin. Often, the packet forwarding diagnostic functions necessary to determine the true source of the attack are not available due to hard-

[1] Even if there were, the lack of validation of the IP headers would probably allow an attacker to bypass such mechanisms by forging packets which would appear to conform to the network usage policy.

ware resource limitations or software implementations which lack the appropriate features.

As if the situation could not possibly get any worse, the amazing success of the Internet has increased the number of Internet hosts which are vulnerable to unauthorized access. This condition makes distributed denial-of-service (DDoS) attacks [7] more feasible than they might have been at one time. In effect, attackers cannot only hide their identity and exploit amplification methods, but also increase the number of hosts used in an attack from a single host to hundreds or thousands of hosts.

This paper presents several different approaches to the problem of tracing the ingress adjacency of reasonably large flows of forged IP packets. It then details a specific solution based on IP tunnels (such as GRE) [8, 9], Border Gateway Protocol (BGP) [10, 11, 12], an Interior Gateway Protocol (IGP) such as Open Systems Interconnect Intermediate System to Intermediate System (IS-IS) [11, 13, 14], and diagnostic features on a subset of routers in the network backbone. Lastly, the benefits and limitations of the method are discussed along with the applicability to DDoS attacks. Methods of preventing such attacks, through source validation for example, are not discussed in this document.

2 Assumptions and Definitions

If two routers are physically or virtually connected and this connection is used to exachange IP packets, then the two routers are considered to be adjacent. The connection between the two routers is referred to as an *adjacency*. *Adjacency capacity* refers to a router's capacity to handle all of the required connections, routing sessions, bandwidth, and anything else needed to maintain a certain number of adjacencies.

In the examples, it is assumed that we are tracking attacks across a hypothetical ISP backbone network. There are two basic classifications of routers used: backbone routers and external routers. *Backbone routers* are routers which are part of the ISP backbone network. *External routers* are routers which are not part of the ISP backbone network; they could belong to a customer or another ISP.

Backbone routers are sub-classified by what they are adjacent to. *Edge routers* are backbone routers that are adjacent to one or more external routers. *Transit routers* are backbone routers which are only adjacent to other backbone routers.

In addition to these routers, we will present a special case of router called a *tracking router* which is conceptually adjacent only to edge routers and other tracking routers. An adjacency between a tracking router and an edge router, or a tracking router and another tracking router, is called a *tracking adjacency*.

Because the example network is an ISP network, it is assumed that attacks originate outside of the network and that they are targeted at a victim outside of the network. Therefore, the malicious packets that comprise the attack will be transmitted across the edge of the network twice: once at the *ingress edge adjacency* (the source of the attack), and once at the *egress edge adjacency* (the destination of the attack), where *edge adjacency* refers to an adjacency between an edge router and an external router. The *ingress edge router* is therefore the edge router which has the ingress edge adjacency, and conversely the *egress edge router* is the edge router which has the egress edge adjacency.

Attack signature refers to some pattern which can be used to help distinguish malicious packets from normal traffic. At the very least, an attack signature is defined by the IP address or address range of the entity that is being attacked. Often it is not possible to determine an attack signature that only matches malicious packets. A good attack signature will predominantly match malicious packets but may also match a certain amount of legitimate traffic.

Input debugging refers to the diagnostic features required to determine from which adjacency a packet matching an attack signature on an individual router arrived. In other words, input debugging is any feature that will reveal which previous hop the attack is coming from or through.

A *tracking hop* is one invocation of input debugging on a particular router. It could also be described as the act of querying a specific router for input debugging information matching a particular attack signature. The number of tracking hops is often referred to in terms of d or d_t where d is the maximum hop diameter of the backbone network and d_t is the hop diameter of the CenterTrack overlay net-

work. There is no fixed relationship between d and d_t since either network could be of an arbitrary hop diameter, but $d_t < d$ for any useful implementation of CenterTrack.

3 Finding a Solution

3.1 Possible Solutions

We originally considered several possible solutions to the problem of tracking forged packets. Several of the ideas that were given consideration are summarized below.

- Hop-by-hop: This is the method used by the DoSTrack[2] script. Input debugging is performed on the edge router closest to the victim in order to determine which adjacency on that router originated the attack packets. Once the adjacency is identified, the process is repeated on that adjacent router. This method is applied recursively until the edge of the network is reached and the edge ingress adjacency is identified. This requires up to d tracking hops.
- Hop-by-hop from center: Traffic for the victim is rerouted to a router at the top level of the network (effectively in the "center" with regard to hop diameter) and then discarded. That router effectively becomes the victim, and hop-by-hop tracking is performed starting with that router. This requires at most $d/2 + 1$ hops. Though it is theoretically unnecessary to discard the traffic in order to perform the diagnostics, it is usually non-trivial or impossible to route the traffic through a particular router and out of the network through the egress adjacency.
- Hop-by-hop through overlay network: This is the method employed by CenterTrack. An overlay network is created which links all edge routers to a central tracking router or a simple network of tracking routers. Dynamic routing is employed which causes only the traffic destined for the victim to be routed through the overlay network. Hop-by-hop tracking is then used, starting with the tracking router closest to the victim. This requires up to $d_t + 1$ tracking hops.
- Traffic flow measurement[3] on edge adjacencies: All edge routers provide some level of traffic flow measurement data on all edge adjacencies. This data must contain, at a minimum, source address, destination address, adjacency, and approximate number of packets. This data is then searched, based on the attack signature, to determine the ingress adjacency of the attack. This requires no tracking hops per se, but it does require a (potentially very large) database search.

Additional areas of research were suggested by other researchers after the original consideration of the problem at UUNET. Two of the more promising ideas, packet marking [15, 16] and ICMP traceback generation [17], are similar to logging traffic flow information except that only a small percentage of the traffic is sampled and traffic flow path information is transmitted to the flow destination instead of a system managed by the network operator. These solutions require significant new protocol development and, while they would provide solutions that are superior to the one discussed in this paper, they are oriented toward a longer-term horizon. A good comparison of various proposals is provided in [16].

3.2 Advantages and Disadvantages

A brief summary of the advantages and disadvantages of each method presented above is described below.

- Hop-by-hop: This method is transparent to the victim and attacker since interrupting the flow of legitimate traffic and alterations to routing are typically unnecessary. This method requires input debugging features on every router

[2] DoSTrack is a Cisco-specific perl script that implements this method. The last official version would not work on any routers configured to use Cisco Express Forwarding, which includes most Cisco routers used by large ISPs today. It is no longer officially distributed or supported by its authors.

[3] See [18] for more information on Traffic Flow Measurement. We are assuming a relatively simple static rule set for this comparison. For an example of a traffic flow measurement system, see the documentation on cflowd [19].

Others are investigating methods which require the configuration of generalized attack signatures on the monitoring systems to reduce the amount of data that is generated and provide for the possibility of active enforcement [20, 21, 22].

in the network. Even if the software functions exist, the hardware resources may not exist especially on multi-gigabit routers operating at full capacity. There are currently no standards for input debugging. This method does not scale well to DDoS attacks due to the large number of operations required.

- Hop-by-hop from center: The primary advantage is that the number of hops required to complete the process is reduced by about 1/2 over the previous method. Otherwise, this method suffers from the same limitations as the previous method. In addition, legitimate traffic usually must be discarded along with the malicious traffic, and routing changes are required.

- Hop-by-hop through overlay network (CenterTrack): With this method, specialized diagnostic features are now required only on edge routers and special-purpose tracking routers. Heavily-loaded high-speed transit routers no longer require such features. Also, the number of hops required to perform tracking is typically reduced to 2 or 3. Because this method requires that an overlay network be created, it increases the complexity of the network and introduces additional administration tasks. Required routing changes may have a global impact on the network if not performed properly; this introduces greater operational risk. While this method is better suited to DDoS attacks than plain hop-by-hop (in terms of number of operations) it is still not ideal. The additional overhead of encapsulating packets is incurred on a larger number of edge routers when used to route a DDoS attack, increasing the potential for collateral damage.

- Traffic flow measurement on edge adjacencies: No network configuration or rerouting is required in order to track malicious flows of traffic; the collected information is simply searched based on the attack signature. Also, because the data are retained for some period of time, the source of an attack can be determined after it has already stopped. All edge routers must be capable of performing the required accounting function, and (in the absence of a manageable, scalable, dynamic configuration method) this function must be performed by all of the edge routers all of the time. The accounting functions generate a tremendous amount of information which consumes network bandwidth. The data must be collected at one location or a small, manageable number of locations in order to be searched effectively. This method does, however, have the best scaling properties with respect to DDoS attacks.

3.3 Design Goals

In choosing and designing a system, the following requirements were taken into account:

- Must be able to identify the source adjacency of the vast majority of DoS floods without interrupting the flow of legitimate traffic.
- Must be as vendor independent as possible, using standard protocols (such as BGP, GRE, etc.) where possible.
- Must utilize existing products or products that will be available in the near future (several months).
- Should limit the number and complexity of features required on transit routers.
- Should introduce as little additional network complexity as possible.
- Should introduce minimal additional operational risk to the backbone.
- The tracking process should be fast and consist of few individual configuration operations.
- Cost and deployment time should be minimal.

It is worth noting that moderate or large scale (more than 10 distributed attackers) DDoS attacks were not a concern when we originally determined our requirements. As a result, scalability to DDoS attacks was not a design goal for this system.

3.4 Other Factors Considered

In addition to the design goals, a number of factors were taken into consideration when deciding which solution to pursue. A few of them are mentioned below.

- Router vendors often do not implement the required software functionality for hop-by-hop

tracking or traffic flow measurement, particularly on multi-gigabit platforms.

- Router vendors typically do not design routers with the necessary resources for packet tracking diagnostic and accounting functions; routers are designed to forward packets rather than analyze them. This is particularly true of the highest-end routers.

- Currently it is impossible in most IP network routing architectures to force traffic for a particular destination through an arbitrary transit router using dynamic routing protocols.

- Dynamic routing processes are often susceptible to being disrupted with unintended announcements or other data which can severely disrupt a network.

- Traffic flow measurement, especially in large networks, typically generates a massive amount of data which is difficult to store and search, requires a massive configuration operation, or requires intelligent meters that can dynamically configure themselves.

3.5 The CenterTrack Decision

We have concluded that the two most promising methods are hop-by-hop tracking through an overlay network and traffic flow measurement. Because the functionality required for hop-by-hop tracking through an overlay network is more readily available and practical in a large network within the desired time frame, we decided to consider that approach first.

4 CenterTrack Design Issues

We have designed a model for an overlay network that we call "CenterTrack," as well as a specific method of implementing it. This model calls for a central tracking system that is built "virtually adjacent" to all edge routers. Edge routers, as well as the equipment that comprises the tracking system, must be able to perform input debugging. A few of the design issues are presented below.

4.1 Tracking Adjacencies

The tracking system must be adjacent to all edge equipment. This can be accomplished using physical connections, layer 2 virtual connections (VCs), or IP tunnels. Physical connections are very cost-prohibitive. Layer 2 VCs are dependent on a particular contiguous layer 2 technology, such as ATM or Frame Relay, which must be accessible by both the tracking system and the edge routers. Design changes which alter the network over which VCs are built may require a rebuild of the tracking network.

IP tunnels can always be built over the existing IP network. Because IP is available to all edge routers, and is not likely to be eliminated from the design of an ISP backbone network, the tunnels can survive underlying network changes. This reduces the number of problems inherent in managing the tracking network.

4.2 Routing Architecture

In order to be practical and effective, some system of dynamic routing must exist between the edge equipment and the tracking system. The system depends on ease of routing updates to edge equipment in order to pinpoint the desired edge router.

At the same time, however, the ability of the overlay network to disrupt normal routing must be reduced as much as possible. A poor implementation could make it too easy for a small error to severely disrupt the network.

An Interior Gateway Protocol, such as IS-IS or OSPF [11, 23], could be used to handle dynamic routing updates between the tracking system and the edge routers, but this would require that the tracking network be treated as an integral part of the backbone network. The lack of administrative distinction between the tracking network and the backbone network would pose a risk to normal operation.

Less risk is involved if a strong administrative distinction exists between the overlay network and the real network. IGPs typically do not allow as much administrative distinction to be made between different parts of the network. Because of this, we decided that designing the network as an external

autonomous system using BGP to handle routing updates would pose less risk while accomplishing the same result.

4.3 Scalability Issues

4.3.1 Single Tracking Router

In a small implementation, the tracking system would consist of a single router with tunnels connecting it to all edge routers. If the number of edge routers is larger than the number of tracking adjacencies serviceable from a single router, multiple routers are necessary.

4.3.2 Single-Level Full Mesh

If we have N_C tracking routers that can each handle C tracking adjacencies then we have CN_C tracking adjacencies available. Since the number of edges required to fully mesh all nodes in an undirected graph is $N_C(N_C-1)/2$, the same number of adjacencies are required to fully mesh all tracking routers. Therefore, this leaves

$$N_E = \left\lceil CN_C - \frac{N_C(N_C-1)}{2} \right\rceil$$

N_E tracking adjacencies available for use by edge routers. Using the quadratic formula to solve for N_C we find that,

$$N_C = \left\lceil (C+\frac{1}{2}) - \sqrt{(C+\frac{1}{2})^2 - 2N_E} \right\rceil$$

for a single level full mesh network of tracking routers, if each tracking router can handle C tracking adjacencies. This means that diminishing returns are observed when adding additional tracking routers until the number of tracking routers reaches a maximum of $\lceil (C-1)/2 \rceil$. After that point, adding additional tracking routers actually reduces the number of available tracking adjacencies for edge routers.

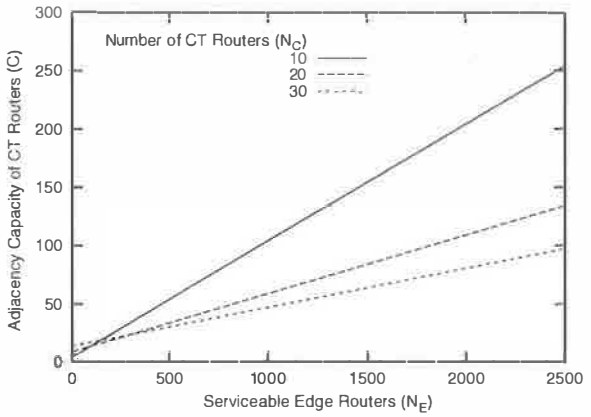


Figure 1: This graph shows how single-level full mesh CenterTrack networks composed of 10, 20, and 30 routers (N_C) scale in terms of edge routers (N_E) with respect to individual tracking router capacity (C).

4.3.3 Two-Level Hierarchy

If we take the full mesh of adjacencies between tracking routers and replace it with a set of adjacencies to a second-level "hub" tracking router, we now have a two-level hierarchy with one router at the top level. In this sort of network, $N_C(N_C-1)/2$ adjacencies between tracking routers and other tracking routers are replaced with N_C-1 adjacencies to a single CenterTrack transit router. This increases the diameter of the CenterTrack network by an additional hop, frees-up $N_C(N_C-1)/2-(N_C-1)$ adjacencies, requires another router, and does not result in much additional edge router capacity(N_E) if N_C is a small percentage of C. If N_C is at its single-level full-mesh maximum of $\lceil (C-1)/2 \rceil$, then moving to this topology effectively doubles the number of tracking adjacencies available for edge routers. In addition, it doubles the maximum possible number of tracking routers. (Achieving maximum capacity requires approximately $C/2$ routers in a single-level full mesh and approximately C routers in a two-level hierarchy.) Figure 2 provides a comparison.

Due to the additional hop introduced, the poor distribution of workload, and the introduction of a single point of failure with this design, it is more desirable to use a single full mesh with a small number of tracking routers that can each handle a large number of tracking adjacencies. The two-level topology is usually only worth considering if the network

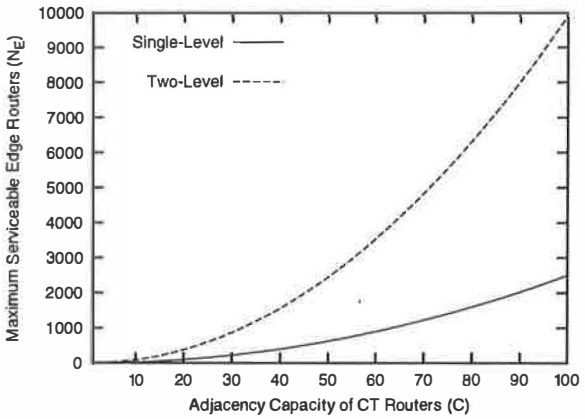


Figure 2: Maximum scalability comparison of different topologies given tracking routers with a capacity of C. This represents the absolute maximum number of edge routers that could be supported.

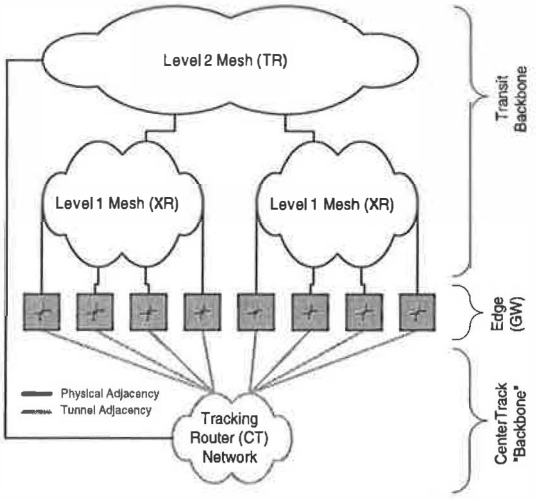


Figure 3: Conceptualized network diagram with two-level backbone.

grows faster than router capacity and N_C is reaching its limit.

4.4 Tracking Router Capabilities

Tracking routers must be able to perform input debugging. They must also be able to handle sufficient tracking adjacencies (using IP tunnels) such that adjacencies can be built between the CenterTrack system and all edge routers without requiring more than a manageable number of tracking routers.

Assuming that you want to manage no more than N_C tracking routers, and you have (or expect to have) N_E edge routers, each tracking router must have a tracking adjacency capacity of:

$$C = \left\lceil \frac{N_C - 1}{2} + \frac{N_E}{N_C} \right\rceil$$

4.5 Edge Router Capabilities

Edge routers also must support input debugging and some acceptable method of IP-over-IP tunneling. Unlike the tracking routers, each edge router is only required to maintain one tunnel.

5 Implementation

5.1 Example Network

The test lab network in Figure 4 is a gross simplification of the current UUNET backbone design. In this network, the "CT" routers are the CenterTrack routers (also referred to as tracking routers) the "TR" and "XR" routers are transit routers, and the "GW" routers are edge routers. The TR routers represent the top level in the transit router hierarchy, and the XRs represent the bottom level. (In a full-scale network, the two TR routers would be a larger full mesh of TR routers and each XR router would be replaced with a regional full mesh of XR routers. See Figure 3.) The CT routers are physically connected at the top level.

An IP tunnel exists between the two CT routers, creating the simplest instance of a full mesh. IP tunnels exist between the CT router in each hub/region and the GW routers in each hub/region. In this way, all edge routers are reachable from each other through the overlay network created by the CT routers.

In BGP routing terms, the backbone routers are in AS 65444, while CT routers are in a separate AS, 65445.

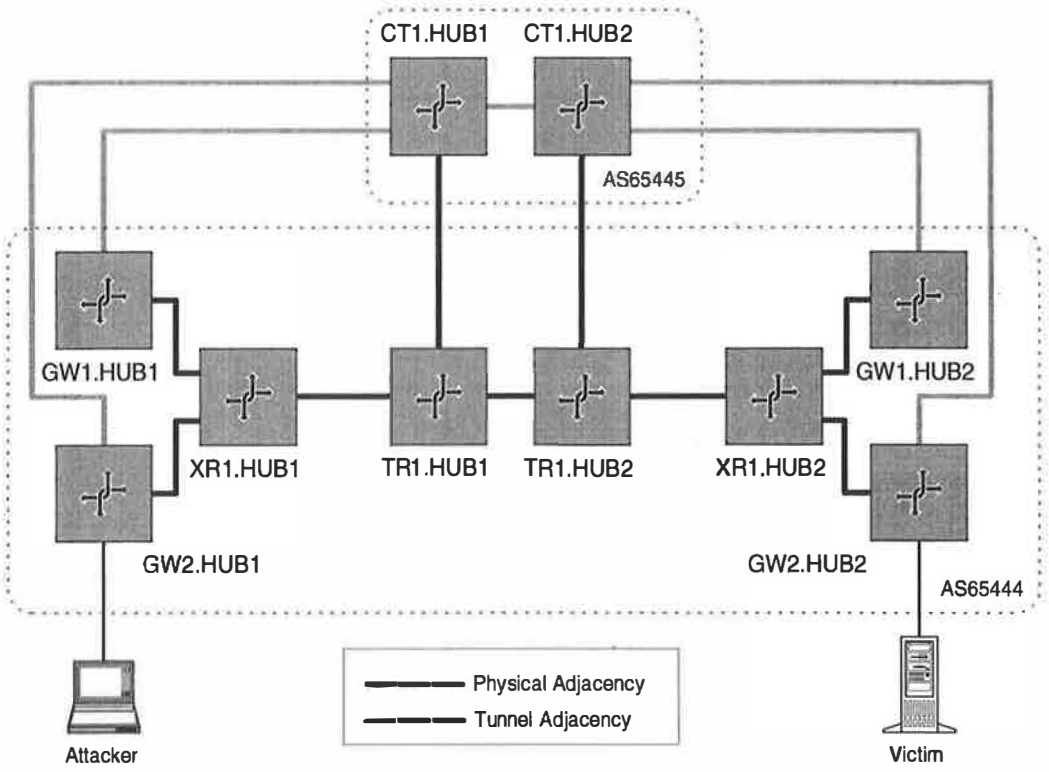


Figure 4: Test Lab Network. This is an extremely simplified version of the UUNET North America backbone. A much larger mesh of TR routers and several separate meshes of XR routers exist in the actual backbone.

5.2 Dynamic Routing With Tunnels

If a router determines, usually due to a routing announcement, that a tunnel's endpoint address is reachable through the tunnel itself, then the tunnel becomes useless. This problem is called *tunnel collapse*. In order to keep this from happening, it is necessary to ensure that no tunnel endpoint address can be announced as being reachable through a tunnel. The easiest way to do this is to number tunnel termination interfaces out of a distinct range of addresses called the *tunnel termination address space*, and use this range to filter the routing announcements. In IP terms, the tunnel termination addresses would be used in the outer IP header of a tunnel packet.

In addition, we do not want traffic that should be using the overlay network to be routed directly out of a tracking router's physical interface without being encapsulated. For this reason, tunnel interfaces should be numbered out of a distinct address range, called the *tunnel interface address space*. (The least-specific, i.e. shortest, prefix for this block should be routed to a discard next-hop to prevent matching a default route.)

With these address ranges defined, the rules can be summarized as follows:

- Tunnel interfaces never announce or accept prefixes from the tunnel termination address space.
- The tracking router's physical interfaces never announce or accept prefixes that are part of the tunnel interface address space.

Configuring the route announcement filtering in this manner will prevent tunnels from collapsing, and will ensure that packets intended to transit the overlay network will not bypass it.

5.3 Tracking Router Physical Connection

Each tracking router must be physically connected to the backbone. It is possible to accomplish the routing part of this with static routing, or EBGP if desired.

In both cases, a primary loopback address is created on the tracking router to serve as an endpoint for tunnels. The primary loopback address must be numbered out of the tunnel termination address space. Also, in both cases, the prefix for the entire tunnel interface address space is routed to a discard next-hop. This is so that any packets destined for an unknown address in that range will not match the default route and be forwarded out of the physical adjacency.

5.3.1 Static Routing

When static routing is employed, the tracking routers have a default route to the backbone through their physical adjacency. The primary loopback is statically routed to the tracking router from the adjacent backbone router.

5.3.2 BGP Routing

If used, BGP should only announce the primary loopback of the tracking router. Only a default route is announced to the tracking router from the backbone.

5.4 Tunnel Termination

Once the tracking routers are physically connected to the network and their primary loopbacks are reachable from the backbone, it becomes possible to create the tunnels. The primary loopbacks of the tracking routers are used as the termination interfaces for tunnels. While physical interface addresses could be used, using the primary loopback interfaces allows for more flexibility with physical interface numbering.

5.5 Tunnel Numbering Methods

On most router implementations, tunnels can be numbered or unnumbered. Using the *numbered* method, /30 blocks out of the tunnel interface address space are assigned to each tunnel. This causes each tunnel interface to have a different IP address. Using the *unnumbered* method, a special-purpose secondary loopback interface is created on each router that will terminate a tunnel. This *overlay loopback* is numbered from the tunnel interface address space. Rather than numbering the tunnels themselves, the routers learn or are configured to know what overlay loopback is reachable through each tunnel.

The numbered method uses more address space and requires approximately as much configuration as the unnumbered method. The unnumbered method uses less address space and is easier to manage because each router only has one next-hop IP address rather than one for every tunnel that terminates on the router. We will assume that the unnumbered method is being used.

5.6 Tracking System IGP and IBGP

Once tunnels have been built such that all of the tracking routers are fully meshed over tunnels, an IGP[4] should be used to distribute link-state information about the tunnels and establish reachability information for the tracking system's overlay loopback interfaces. The IGP should only advertise the overlay loopback addresses. In particular, if the primary loopbacks are advertised then tunnels will collapse.

Because EBGP will be used by all tracking routers to announce routes to the edge routers, all tracking routers must have IBGP sessions established with each other. Filters must be applied to prevent the primary loopback addresses from being learned through IBGP. If filters are not applied, then the tunnels will collapse.

[4] In our test implementation, IS-IS was used for this.

5.7 Edge Tunnels

Edge tunnels between the edge routers and the tracking routers are built in the same manner as the internal tunnels used to mesh the tracking routers together. An overlay loopback is created on each edge router, and a static route for the tracking router overlay loopback is added to establish reachability of the tracking router through the tunnel. Similarly, a static route is added on the tracking router which establishes reachability of the edge router's overlay loopback through the tunnel. In this manner, connectivity is established which can be used to establish an EBGP session.

The static routes were unnecessary on the internal tunnels because an IGP was being used. An IGP is not used with the edge routers to avoid conflict with the backbone IGP.

5.8 Edge Tunnel EBGP Sessions

EBGP is configured, using the overlay loopback addresses, between each edge router and the corresponding tracking router. The following filters are used:

1. The edge routers are configured:
   (a) to accept all prefixes from the tracking system that are not within the tunnel termination address space.
   (b) to set the BGP local-preference attribute high so that the additional AS hop introduced into the path will not cause the route to be ignored in favor of a competing route with the same prefix length.
   (c) not to announce any prefixes to the tracking system.
2. The tracking routers are configured:
   (a) to ignore all prefixes originating from any edge routers.
   (b) to only advertise local prefixes (except those in the tunnel termination address space) to the edge routers.

6 Usage

6.1 Static Routes

To use CenterTrack, the packets to be tracked must be flowing over the CenterTrack network. To accomplish this, static routes are added on two routers.

- A static route[5] for the victim, pointing through the egress edge adjacency, is added on the egress edge router. This ensures that a more specific route learned from CenterTrack will not take precedence over the normal egress path.
- A static route for the victim, pointing through the tunnel to the egress edge router, is added on the tracking router adjacent to the egress edge router. This route is announced via IBGP to the other tracking routers, and then announced to all edge routers via EBGP.

Once routing converges, all[6] traffic for the victim will take a path through the overlay network.

It is important that all traffic continue to be routed to the victim because the victim generally still wants to receive legitimate traffic. Due to limitations of most IP forwarding implementations and dynamic routing protocols, it is not possible to only reroute the traffic matching a complex attack signature. Therefore all traffic for a victim must be rerouted through the overlay network, and all the complex pattern matching against an attack signature must be done in the input debugging process. Filters might also be placed on the tracking routers to deny passage to the attack packets if the edge routers are unable to perform the required filtering.

6.2 Hop-by-Hop Tracking

Once traffic has been rerouted, it is possible to employ hop-by-hop tracking to find the source of the attack. The process for tracking an attack from a single source is described below. (This paper does

[5] Arbitrary length prefix, usually a host route.

[6] Well, not quite. Traffic entering the network at the egress edge router will never be transmitted across the CenterTrack network. In other words, if the egress router is also the ingress router then the traffic will never be routed over the overlay network.

not address issues related to automating this process in software.)

1. The initial starting point is the tracking router closest to the victim.
2. Input debugging is performed on the current router.
   (a) If no attack is seen from any of the tunnels, then the attack must be originating from something (another customer or peer) adjacent to the egress edge router. Input debugging is then performed on the egress edge router to find the source. The process ends.
   (b) If an adjacency is identified as the source of the attack, then
      i. If the adjacency connects to a backbone router or tracking router, repeat step 2 on the corresponding router. (This router becomes the "current router.")
      ii. If the adjacency connects to an external router, we have found the ingress adjacency and the process ends. Filters can be applied, the attacking site contacted, etc.

An attack from multiple sources can also be tracked using this method; where multiple source adjacencies are identified, each one must be investigated separately (possibly in parallel) until multiple edge adjacencies are identified. This capability is useful since multiple source attacks are quite common; attackers often "team up" to tackle a single victim, essentially creating a small-scale "manual" DDoS attack. However, this method is not scalable with regards to DDoS attacks in general unless the method is highly automated.

6.3 Packet Capture

Full packet capture can be useful for analyzing a new attack in detail and recording evidence of an attack for use in prosecution. Even though most routers are not capable of full packet capture, it is still possible to use this system to help "sniff" most traffic, for a specific destination, that enters the backbone. This could be accomplished by using systems capable of full packet capture as the tracking routers,

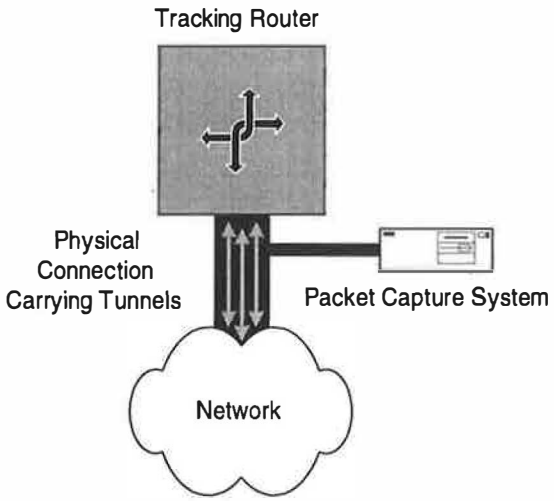


Figure 5: Using a sniffer in conjunction with a tracking router. This would be useful when the tracking router cannot perform all of the desired traffic analysis functions.

or by attaching specialized sniffers on the physical connections between the tracking routers and the backbone, as show in Figure 5. It cannot be guaranteed that all traffic for a given destination can be sniffed in this manner because, if the source and destination are connected to the same edge router, traffic will not transit the tracking network.

Aside from packet capture, a separate system could also be used to perform the input debugging function if, for example, a particular tracking router is desirable for its ability to perform IP encapsulation but lacking in diagnostic functions. Effectively this would distribute the tracking router functionality between two machines.

7 Problems and Limitations

7.1 Attacks from Within

This system is not very useful for tracking attacks that originate from within the backbone itself. However, an attack that originates from a transit router that is directly connected to an edge router is equivalent to an attack that originates from outside of

the backbone, and it can be tracked just as easily. This system does not simplify the process of tracking attacks originating from within the backbone and more than one hop away from an edge router.

7.2 Attacks on Backbone Routers

It is difficult or impossible to use this system to help track an attack in which a backbone router is a victim. Attempting to reroute traffic destined to a backbone router interface over the tracking network would cause tunnel collapse or routing loops.

7.3 "Stepping Stone" Routers

An attacker can bypass the overlay network by "bouncing" the attack off of one of the transit routers. For example, an attacker sends ICMP echo request packets to TR1.HUB1 and forges the source address as the address of a victim.

7.4 Tunnel Overhead

Assuming the attacker were to use the smallest possible IP packet with a 20 byte header and an empty payload, the additional IP header used to encapsulate the packet would increase the packet size by over 100%. Though the bandwidth consumed by the attack at the edge adjacency would not be affected, this would effectively double the bandwidth utilization of the attack as it transited the backbone in encapsulated form. Therefore, the overlay network can be exploited to amplify the effects of the DoS attack on the backbone itself.

One possible solution to this is some sort of asymmetric load sharing of the rerouted flow between the overlay network and the physical backbone. This might allow, for example, 10% of the desired packet flow to be transmitted over the CenterTrack network while the rest is transmitted normally over the backbone. We have not yet thoroughly investigated methods for accomplishing this.

7.5 Tunnel Authentication

If the tunnels are not authenticated in some way, attackers can exploit the overlay network to further conceal the source of their attack. With some easily guessable knowledge of the tunnel endpoints, the tunnel packets themselves can be forged, effectively inserting arbitrary packets into a tunnel from afar. While some weak authentication features of GRE might help with this, they have been removed from the latest version of the RFC [8, 9] and are unimplemented in many newer routers. Using the IPSEC authentication header in tunnel mode [24, 25] provides a solution to this problem, but IPSEC is not widely implemented on general purpose high-capacity IP routing equipment at the current time. Using layer 2 VCs can also solve this problem, but such connections suffer from previously stated disadvantages.

7.6 Visibility

Because all traffic to the victim is rerouted over an unusual path, the attacker could detect that the system is being used by using traceroute or a similar tool. This visibility can be limited by configuring the tracking routers in some manner so that they will not send ICMP TTL Exceeded messages. If this is done, the attacker may notice that something has changed, but will have less information about exactly what is going on. It is also possible to have the tracking routers respond with bogus TTL Exceeded messages in order to misrepresent the actual network path.

7.7 Distributed DoS Attacks

It is not obvious how scalable this approach is with regards to large DDoS attacks. The number of operations required to track-down all of the ingress points is slightly larger than the number of ingress points, which could be very large. Also, the overhead of tunnel encapsulation could dangerously amplify the effects of a DDoS attack on the network backbone.

8 Conclusions

The hop-by-hop tracking method employing input debugging is basically sound. However, a simplified tracking network connecting all edge routers via a small number of hops provides an optimized environment for using hop-by-hop tracking. Furthermore, building the tracking network as an overlay network, using IP tunnels, allows the tracking network to survive changes to the backbone architecture at a minimal cost of increased complexity and bandwidth usage.

While a single tracking router may be sufficient for small networks, a single level fully-meshed network of tracking routers is required for large ISP backbones. A two-level system can be used, but the extra hop that is introduced generally outweighs the scaling benefits.

In addition to tracking forged packets, the system can be used for full packet capture provided that the necessary capabilities exist at the tracking routers. This is useful for detailed analysis of attacks.

There are a number of weaknesses in the system which limit its applicability and effectiveness in many situations. However, many of these problems have yet to be overcome in practice. More work is needed to determine the usefulness of this approach.

9 Experiences

CenterTrack has been successful in lab testing but has not yet been implemented in production. We are planning to deploy such a system at some point in the future, however.

10 Acknowledgments

This work was funded by UUNET Technologies, Inc. The author wishes to thank Matthew Sibley for the original idea, and Mark Krause for supporting the effort. Contributions by the following people were greatly appreciated: Eric Brandwine, Clarissa Cook, Ken Dahl, Todd MacDermid, Vijay Gill, Keith Howell, Robert Noland.

References

[1] Steven M. Bellovin. Security Problems in the TCP/IP Protocol Suite. Computer Communications Review, 9(2):32-48, April 1989.

[2] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing. RFC 2267, January 1998.

[3] Computer Emergency Response Team. CERT Advisory CA-96.26: Denial-of-Service Attack via pings. http://www.cert.org/advisories/CA-96.26.ping.html, December 1996.

[4] Computer Emergency Response Team. CERT Advisory CA-98.01: "smurf" IP Denial-of-Service Attacks. http://www.cert.org/advisories/CA-98.01.smurf.html, January, 1998.

[5] Craig A. Huegen. The Latest in Denial of Service Attacks: "Smurfing." http://users.quadrunner.com/chuegen/smurf.cgi. February, 2000.

[6] Computer Emergency Response Team. CERT Advisory CA-96.21: TCP SYN Flooding and IP Spoofing Attacks. http://www.cert.org/advisories/CA-96.21.tcp_syn_flooding.html. August, 1998.

[7] Computer Emergency Response Team. Results of the Distributed-Systems Intruder Tools Workshop. http://www.cert.org/reports/dsit_workshop-final.html. November 1999.

[8] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation. RFC 1701, October 1994.

[9] S. Hanks, T. Li, D. Farinacci, D. Meyer, and P. Traina. Generic Routing Encapsulation. RFC 2784, March 2000.

[10] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 1771, March 1995.

[11] Christian Huitema. *Routing in the Internet.* Prentice Hall, 1995.

[12] Bassam Halabi. *Internet Routing Architectures.* New Rider's Publishing, 1997.

[13] Intermediate system to Intermediate system intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode Network Service. ISO DP 10589, International Standards Organization, 1992.

[14] R. Callon. Use of OSI IS-IS for Routing in TCP/IP and Dual Environments. RFC 1195, December 1990.

[15] Hal Burch and Bill Cheswick. Tracing Anonymous Packets to their Approximate Source. Unpublished paper, December 1999.

[16] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical Network Support for IP Traceback. To appear in SIGCOMM 2000, Stockholm, Sweden, July 2000.

[17] Steven M. Bellovin. ICMP Traceback Messages. IETF Internet Draft, draft-bellovin-itrace-00.txt, March 2000. (Expires September 2000.)

[18] N. Brownlee, C. Mills, and G. Ruth. Traffic Flow Measurement: Architecture. RFC 2063, January 1997.

[19] *cflowd.* http://www.caida.org/Tools/Cflowd/

[20] Glenn Sager. Security Management in Next Generation Networks, a presentation. PICS, July 1998.
http://www.caida.org/NGI/Security/0798/

[21] Glenn Sager. Security Fun with OCxmon and cflowd, a presentation. PICS, at the Internet2 Working Group meeting, November 1998.
http://www.caida.org/NGI/Security/1198/

[22] J. Apisdorf, k. claffy [sic] (NLANR), and K. Thompson. OC3MON: Flexible, Affordable, High-Performance Statistics Collection. MCI/vBNS and NLANR, Internet Society INET '97, January 1997.

[23] J. Moy. OSPF Version 2. RFC 2328, April 1998.

[24] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[25] S. Kent and R. Atkinson. IP Authentication Header. RFC 2402, November 1998.

A Multi-Layer IPsec Protocol

Yongguang Zhang Bikramjit Singh
HRL Laboratories, LLC
{ygz,bsingh}@hrl.com

Abstract

IPsec [KA98c] is a suite of standard protocols that provides security services for Internet communications. It protects the entire IP datagram in an "end-to-end" fashion; no intermediate network node in the public Internet can access or modify any information above the IP layer in an IPsec-protected packet. However, recent advances in internet technology introduce a rich new set of services and applications, like traffic engineering, TCP performance enhancements, or transparent proxying and caching, all of which require intermediate network nodes to access a certain part of an IP datagram, usually the upper layer protocol information, to perform flow classification, constraint-based routing, or other customized processing. This is in direct conflict with the IPsec mechanisms. In this research, we propose a multi-layer security protection scheme for IPsec, which uses a finer-grain access control to allow trusted intermediate routers to read and write selected portions of IP datagrams (usually the headers) in a secure and controlled manner.

1 Introduction

The Internet community has developed a mechanism called *IPsec* for providing secure communications over the public Internet. IPsec can provide data integrity, origin authentication, data confidentiality, access control, partial sequence integrity, and limited traffic flow confidentiality services for communications between any two networks or hosts [KA98c]. By addressing the security issues at the IP layer and rendering the security services in a transparent manner, IPsec attempts to relieve software developers from the need to implement security mechanisms at different layers or for different Internet applications. Arguably, IPsec is the best available mechanism for Virtual Private Networks (VPN) and secure remote accesses.

1.1 The Protection Model in IPsec

The fundamental concept behind the IPsec technology is as follows. The path between an IP datagram's source and destination is divided into three segments (see Figure 1) — the protected and trustworthy local network at the source (e.g., a company's private LAN), the untrustworthy public Internet segment, and the protected and trustworthy local network at the destination. The IPsec architecture places a security gateway (here G_1 and G_2) at each boundary between a trustworthy and an untrustworthy network. Initially, G_1 at the source establishes a security association with G_2 on the destination side, which is a security relationship that involves negotiation of security services and shared secrets. Before an IP datagram (from S to D) is sent to the untrustworthy Internet, the security gateway (G_1) encrypts and/or signs the datagram using an IPsec protocol. When it reaches the security gateway at the destination side (G_2), the datagram is decrypted and/or checked for authentication, before it is forwarded to the destination (D). In some cases, the trustworthy local network on either side can be omitted, and the source or destination host can perform encryption, authentication and other security-gateway functions itself.

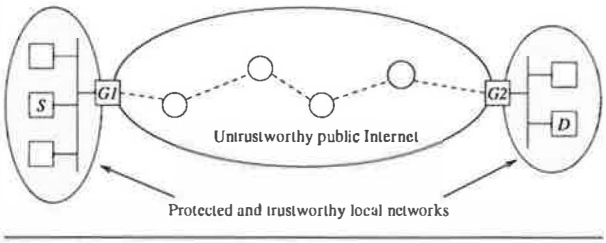


Figure 1: System Model

The IPsec architecture uses two protocols to provide traffic security – AH (Authentication Header) [KA98a] and ESP (Encapsulating Security Payload) [KA98b]. AH provides integrity and authentication without confidentiality; ESP provides

confidentiality, with optional integrity and authentication. Each protocol supports two modes of use: *transport mode* and *tunnel mode*. Transport mode provides protection primarily for upper layer protocols, while in tunnel mode the protection applies to the entire IP datagram.

The granularity of security protection in the IPsec architecture is at the datagram level. It treats everything in an IP datagram after the IP header as one integral unit. Usually, an IP datagram has three consecutive parts – the IP header (for routing purposes only), the upper layer protocol headers (for example, the TCP header), and the user data (for example, the TCP data). In transport mode, an IPsec protocol header (AH or ESP) is inserted in after the IP header and before the upper layer protocol header to protect the upper layer protocols and user data. In tunnel mode, the entire IP datagram is encapsulated in a new IPsec packet (a new IP header followed by an AH or ESP header). In either case, the upper layer protocol headers and data in an IP datagram are protected as one indivisible unit (see Figure 2).

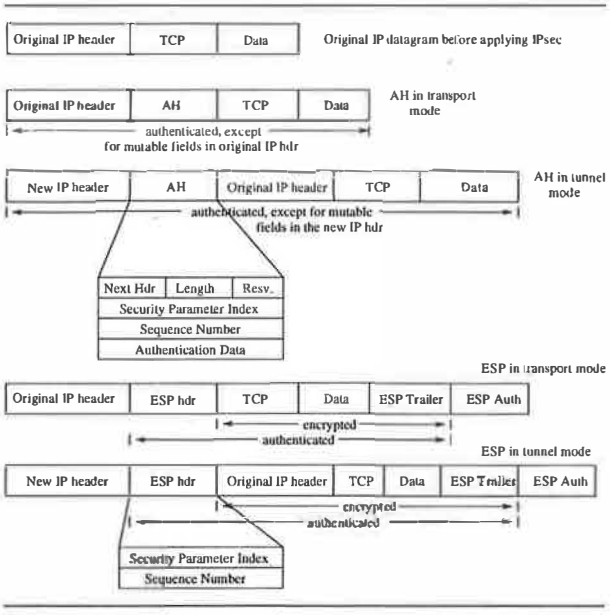


Figure 2: The Protocol Formats of IPsec-protected IPv4 Packets (assuming TCP)

The keys used in encryption and authentication are shared only by the sender-side and receiver-side security gateways. All other nodes in the public Internet, whether they are legitimate routers or malicious eavesdroppers, see only the IP header and will not be able to decrypt the content, nor can they tamper with it without being detected. Traditionally, the intermediate routers do only one thing – forward packets based on the IP header (mainly the destination address field); IPsec's "end-to-end" model is well-suited to this layering paradigm.

1.2 Limitations of End-to-End Security

However, this protection model and its strict layering principle are unsuitable for an emerging class of new networking services and applications for the next generation Internet. Unlike in the traditional minimalist Internet, intermediate routers begin to play more and more active roles. They often rely on some information about the IP datagram payload, such as certain upper layer protocol header fields, to make sophisticated routing decisions. In other words, routers can now participate in a layer above the IP. Examples of such active networking techniques are:

- Internet traffic engineering. The Internet is moving towards active traffic engineering to meet increasing demand for bandwidth and rich services. Routers/switches will support per-flow and class-based queueing to give fair bandwidth access to all users. A QoS guarantee will be provided to traffic flows generated by paying customers. Router-based congestion control mechanisms, such as Random Early Detection (RED) [FJ93] with penalty box [FF99], also require intermediate nodes to discriminate between traffic flows. Depending on the granularity used in defining a "flow," certain nodes in the middle of the network may need access to information in the upper layer protocols, such as TCP/UDP port numbers, to classify packets into flows before applying discriminating operations.

- Transport-aware link layer mechanisms. The global Internet has accommodated a very wide range of link technologies, but certain transport protocols like TCP have not achieved optimal performance when operated over a path that includes lossy wireless links or long-delay satellite links. For example, in a recent paper [BPSK97], Balakrishnan proves that, to significantly improve the TCP's performance over a wireless link, the base station at the lossy link must be aware of the TCP state information in each passing flow, and deliberately delay or drop certain types of TCP packets. Such link-layer

mechanisms for TCP performance improvement (often referred to as *TCP Performance Enhancing Proxies* or TCPPEP [BKGM00]) require intermediate nodes to access and sometimes modify the upper layer protocol headers.

- Application-layer proxies/agents. Some Internet routers can provide application-layer services for performance gains. For example, an intermediate router can become a transparent web proxy when it snoops through the TCP and HTTP header of a bypassing IP datagram to determine the URL request, and serves it with the web page from the local cache. It is transparent to end-users but boosts the responsiveness because the delivery paths for web requests and data between the intermediate router and the web site server are eliminated.

- Active networks. Going one step further, the active network architecture is a new networking paradigm in which the routers perform customized computation on the data flowing through them. A number of experimental active network systems have been developed and they can be run over the Internet. In this architecture, a single IP datagram carries not only upper-layer protocol headers and user data, but also a "method" – a set of executable instructions to be interpreted by the intermediate routers, for describing, provisioning, or tailoring network resources and services in order to achieve the delivery and management requirements. Obviously then, the "method" portion of the IP datagram ought not to be encrypted "end-to-end."

- Traffic Analysis. Many network operators actively monitor the traffic for accounting or for intrusion detection purposes. Usually, such monitoring requires logging of certain upper layer protocol information, like TCP/UDP ports. Many firewalls that protect local networks also depend on such information to deny unauthorized traffic.

All these mechanisms require intermediate network nodes to access information encoded in the IP datagram payload, but the current IPsec technology advocates end-to-end security and prevents such access. This fundamental conflict [NBB99] makes it a very difficult problem to provide both security and extensibility in one unified platform.

1.3 Problem Statement

The goal of this research is to develop a security scheme that supports the above new network services and applications under the IPsec framework. The new scheme should *grant trusted intermediate routers a secure, controlled, and limited access to a selected portion of certain IP datagram, while preserving the end-to-end security protection to user data.*

2 Approaches

We have investigated three ways to solve the problem – replacing IPsec with a transport-layer security mechanism, using a transport-friendly ESP format, and developing a multi-layer protection model for IPsec.

The first approach, replacing IPsec with a *transport-layer mechanism*, circumvents the problem of intermediate nodes not being able to access the encrypted TCP headers, yet introduces certain other difficulties. There are actually several transport-layer security mechanisms available today, including SSL (most notably used in Netscape and other WWW applications) or TLS (a proposed IETF standard [DA99]). Both SSL and TLS encrypt the TCP data while leaving the TCP header in unencrypted and unauthenticated form so that intermediate nodes can make use of the TCP state information encoded in the TCP header. However, letting the entire TCP header appear in clear text exposes several vulnerabilities of the TCP session to a variety of TCP protocol attacks (in particular traffic analysis), because the identity of sender and receiver are now visible without confidentiality protection.

Alternatively, it is possible to tunnel one security protocol within another, such as SSL/TLS inside an IPsec ESP – letting SSL/TLS protect the TCP data and ESP protect the TCP header. However, there is a problem here too because ESP encrypts both TCP header and TCP payload (SSL/TLS-protected data) as a whole. Thus, the encryption/authentication/decryption has to be done twice on the TCP data part, an unnecessary waste of resources. The intermediate router, for example, must decrypt the entire packet to access just the TCP header information.

The second approach is to develop a *transport-friendly ESP* (TF-ESP) protocol format for IPsec. Proposed by Steve Bellovin of AT&T Labs [Bel99], TF-ESP modifies the original ESP protocol to include limited TCP state information, such as flow identifications and sequence numbers, in a disclosure header outside the encryption scope (but authenticated). This approach will work well for some TCP PEP mechanisms such as TCPPEP for wireless network (e.g., TCP snooping), but it may not suite other mechanisms that need a write access, such as TCPPEP for satellite networks [ZDRD97, BKGM00]. To support TCPPEP for satellite networks, the TCP state information also needs to be placed outside the authentication scope. Without proper integrity protection, this can be dangerous. Further, the unencrypted TCP state information is made available universally, including to untrustworthy nodes, which creates vulnerability for possible attacks. In addition, TF-ESP is not flexible enough to support all upper-layer protocols.

Since the above two approaches both have limitations, we thus propose a third approach – to develop a *multi-layer security protection scheme* for IPsec. The idea is to divide the IP datagram into several parts and apply different forms of protection to different parts. For example, the TCP payload part can be protected between two end points while the TCP/IP header part can be protected but accessible to two end points plus certain routers in the network. The rest of this paper will describe the principle, the design and an implementation of this approach.

3 The Principle of Multi-Layer Security Protection

Our approach is called ML-IPsec (Multi-Layer IPsec). It uses a multi-layer protection model to replace the single end-to-end model. Unlike IPsec where the scope of encryption and authentication apply to the entire IP datagram payload (sometimes IP header as well), our scheme divides the IP datagram into zones. It applies different protection schemes to different zones. Each zone has its own sets of security associations, its own set of private keys (secrets) that are not shared with other zones, and its own sets of access control rules (defining which nodes in the network have access to the zone).

When ML-IPsec protects a traffic stream from its source to its destination, the first IPsec gateway (or source) will re-arrange the IP datagram into zones and apply cryptographic protections. When the ML-IPsec protected datagram flows through an authorized intermediate gateway, a certain part of the datagram may be decrypted and/or modified and re-encrypted, but the other parts will not be compromised. When the packet reaches the last IPsec gateway (or destination), ML-IPsec will be able to reconstruct the original datagram. ML-IPsec defines a complex security relationship that involves both the sender and the receiver of a security service, but also selected intermediate nodes along the traffic stream.

For example, a TCP flow that desires link-layer support from the network can divide the IP datagram payload into two zones: TCP header and TCP data. The TCP data part can use an end-to-end protection with keys shared only between the source and the destination (hosts or security gateways). The TCP header part can use a separate protection scheme with keys shared among the source, the destination, and certain trusted intermediate node. (See Figure 3.) This way, no one in the public Internet other than the source, the destination and the trusted intermediate nodes has access to TCP header or TCP data, and no one other than source and destination (not even the trusted intermediate node) has access to TCP data.

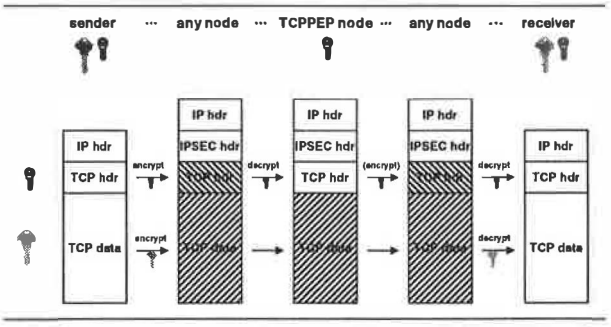


Figure 3: Multi-Layer Protection Model for TCP

This scheme in effect provides a finer-grain access control to the IP datagram. Since ML-IPsec allows network operators and service providers to grant intermediate nodes limited access to IP datagram contents parts (such as TCP header), such access must be granted in a secure and controllable way. The identity of the intermediate nodes must be authenticated (using an out-of-band mechanism such as a public-key infrastructure) to prevent any man-in-the-middle attack. After authentication, keys or

shared secrets corresponding to the authorized IP datagram zones must be distributed to the intermediate nodes, also using out-of-band mechanisms like IKE [HC98].

4 ML-IPsec Design Details

The architecture of ML-IPsec embraces the notion of zones, a new type of security association, the new AH and ESP header formats, and the inbound/outbound processing of ML-IPsec protocol packets. It is designed to be fully compatible with the original IPsec in both protocol formats and processing software.

4.1 Zones

A zone is any portion of IP datagram under the same security protection scheme. The granularity of a zone is 1 octet. The entire IP datagram is covered by zones, except for the IP header in the transport modes, but zones cannot overlap. Using the same TCP example, the portion of the IP datagram that contains TCP header (21st to 40th octet) is Zone 1, and the TCP data portion (41st and above octet) is Zone 2 (assuming transport mode and no TCP options).

A zone need not be a continuous block in an IP datagram, but each continuous block is called a *subzone*. A *zone map* is a mapping relationship from octets of the IP datagram to the associated zones for each octet. Figure 4 below shows a sample zone map.

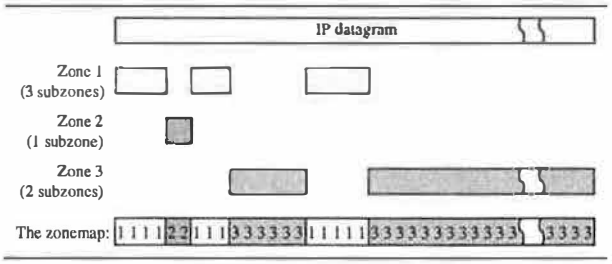


Figure 4: A Sample Zone Map

The zone map is a constant in a security relationship. That is, the zone boundaries in each IP datagram must remain fixed in the lifetime of the security association; otherwise, it will be extremely difficult to do zone-by-zone decryption and authentication. Since IP datagrams are variable in length, the zone that covers the last part of the datagram, usually the user data, should also be variable in size. Zone 3 of the above is an example. It is also possible, theoretically, to define a phantom zone that does not correspond to any byte in an IP datagram.

4.2 Security Association (SA)

4.2.1 Original SA for IPsec

Security Association (SA) is a key concept in the IPsec technology [KA98c]. It is a one-way relationship between a sender and a receiver that affords security services. Each SA defines a set of parameters including the sequence number and anti-replay window for anti-replay service, the protocol mode (transport or tunnel), the lifetime of the SA, the path MTU and other implementation details. For authentication services in AH or ESP, each SA also defines the choice of cryptographic algorithm, the crypto-keys, key lifetimes and related parameters. For encryption services in ESP, each SA further defines the choice of encryption algorithm, the encryption keys, the initial values, key lifetimes, etc. When an outbound IP datagram passes the security gateway, the IPsec module first compares the values of the appropriate fields in the IP datagram (the selector fields) against a set of predefined policies, called SA selectors, in the Security Policy Database (SPD). It then determines the SA for this datagram if any, and does the required security processing (e.g., encryption). When an inbound IPsec datagram passes the security gateway, the IPsec module uses the SPI (Security Parameter Index) field to determine the SA for this datagram and performs security processing (e.g., decryption). Figure 5 gives a simple illustration of how these pieces are connected together in the IPsec architecture.

4.2.2 Composite SA for ML-IPsec

SA in the original IPsec defines a simple security relationship from the sender to the receiver that affords the protection service. ML-IPsec however requires a much more complex security relationship to include sender and receiver, as well as the selected intermediate nodes. Since the security service is zone-by-zone, conceptually we can use an indi-

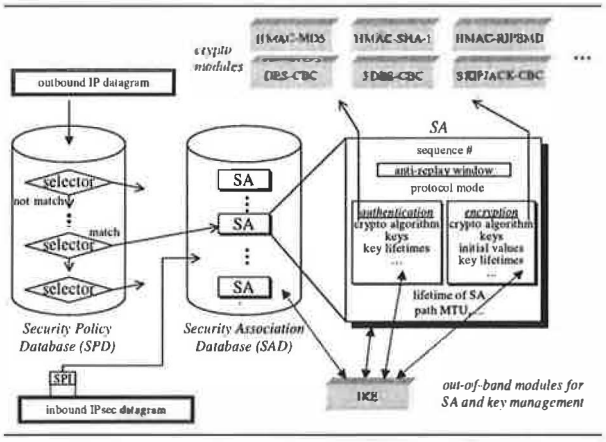


Figure 5: IPsec System Architecture

vidual security relationship to cover each zone, and then build a composite relationship to cover the entire IP datagram. Mapping this idea to the basic Security Association (SA) concept, ML-IPsec needs a new type of SA called *Composite SA* (CSA). CSA is a collection of SAs that collectively afford a multi-layer security protection for the traffic stream.

A CSA has two elements. The first element is a zone map. The zone map specifies the coverage of each zone in an IP datagram. The zone map must be consistent in all nodes involved in the same ML-IPsec relationship. The second element in a CSA is a *zone list*. A zone list is a list of SAs for all the zones. Each and every such SA is stored in the Security Association Database (SAD) [KA98c]. However, some of the fields are used differently in ML-IPsec than as defined in the original IPsec [KA98c]. The following SAD fields, for example, are applicable only on the corresponding zone of the SA.

- Lifetime of this Security Association
- AH Authentication algorithm, keys, etc.
- ESP Encryption algorithm, keys, IV mode, IV, etc.
- ESP Authentication algorithm, keys, etc.

The other SAD fields have no meanings on the zone level. With the exception of a *designated SA* in the zone list, the following SAD fields are not used in other zonal SAs, although they may be initialized during the SA creation process.

- Sequence Number Counter
- Sequence Counter Overflow
- Anti-Replay Window
- IPsec protocol mode
- Path MTU

The designated SA however operates on these fields as defined in the original IPsec. The designated SA is a special SA in the zone list, usually the first SA in the list. It is responsible for maintaining parameters for the IP datagram layer and "represents" the CSA in security processing.

The zone map and zone list can be stored with the designated SA as additional fields in the SAD, or, they can be stored in a separate CSA database. This is an implementation choice and it allows flexibility in adding ML-IPsec features to an existing IPsec implementation.

On inbound processing, if the traffic stream is under ML-IPsec protection, the destination IP address, the IPsec protocol type, and the SPI identifies an entry in the SAD, which points to the designated SA of the CSA for this traffic stream. Or, under alternative implementation, the triplet identifies an entry in the CSA database. By traversing CSA's zone list, ML-IPsec can further identify the SA entries for all the zones.

On outbound processing, the Security Policy Database (SPD) [KA98c] will have a pointer to the designated SA or an entry in the CSA database. Just as in the original IPsec, the selectors will direct the outbound traffic to the proper SPD entry.

4.2.3 Access Control in a CSA

A CSA involves the sender, receiver, and all the authorized intermediate nodes that collectively provide a multi-layer security protection for a traffic stream. Therefore, an instance of CSA must be created in each of these nodes before the ML-IPsec service can commence. It will have these features. First, the zone map must be distributed and remain the same for all nodes. Second, each CSA instance must have a designated SA, and the choice of designated SA must be consistent across all the nodes. Finally, because the designated SA is the one and only SA responsible for the integrity of the IPsec

header (with fields like SPI and sequence number), all these nodes must be able to process this SA.

However, the zone list need not be the same for all nodes. In principle, each zonal SA independently determines the access list for that zone and not all nodes will have access to all zones. If some node does not have access to a zone, the corresponding zonal SA in the zone list will be null. For a particular zonal SA, an instance must be created in each authorized node and stored in its SAD as a step in CSA creation. By determining which zonal SA is to be created in which node, CSA enforces a multi-layer access control for an IP traffic stream.

Since the designated SA must be consistent across all nodes involved in a CSA, they should all have access to the corresponding zone. For convenience, we call this zone for which the designated SA is chosen the *designated zone.* The requirement that all nodes must have access to one common zone is very natural in most applications; the designated zone is usually the first zone in the list, containing the IP header plus certain upper protocol headers. In rare cases where the zones accessible by intermediate nodes are disjoint, we must define a phantom zone of zero size and make it the designated zone. We can now make an SA for this zone and use it as the designated zone. This however introduces extra overhead because the protocol needs to accommodate one more SA.

4.2.4 A TCP Example

Here is an example to illustrate the concept of CSA. It is a traffic flow from Sender (the ultimate source or the outbound IPsec gateway) to Receiver (the ultimate destination or the inbound IPsec gateway), passing through Gateway (an intermediate router providing diffserv or TCPPEP service). Let's assume the desired security service is ESP transport mode.

The corresponding CSA in Sender or Receiver will have the following elements:

- zone map
  - zone 1 = byte 1-20
  - zone 2 = byte 21-EOP (*end-of-packet*)
- zone list
  - SA1 (designated)
    - * sequence number counter
    - * sequence counter overflow
    - * anti-replay window
    - * protocol mode = TRANSPORT
    - * path mtu
    - * lifetime
    - * ...
    - * encryption algorithm = DES-CBC
    - * encryption key = *key1*
    - * authentication algorithm = HMAC-MD5-32
    - * authentication key = *key2*
    - * ...
  - SA2
    - * ...
    - * lifetime
    - * ...
    - * encryption algorithm = 3DES-CBC
    - * encryption key = *key3*
    - * authentication algorithm = HMAC-MD5-96
    - * authentication key = *key4*
    - * ...

The corresponding CSA in Gateway will have the following elements:

- zone map
  - zone 1 = byte 1-20
  - zone 2 = byte 21-EOP
- zone list
  - SA1 (designated)
    - * sequence number counter
    - * sequence counter overflow
    - * anti-replay window
    - * protocol mode = TRANSPORT
    - * path mtu
    - * lifetime
    - * ...
    - * encryption algorithm = DES-CBC
    - * encryption key = *key1*
    - * authentication algorithm = HMAC-MD5-32
    - * authentication key = *key2*
    - * ...
  - SA2 = NULL

Here HMAC-MD5-32 is a hypothetical keyed hash algorithm that produces a smaller 4-octet signature. Using smaller size authentication data on certain zones (usually the protocol headers) has the advantage of lower overhead. Otherwise, the standard HMAC-MD5-96 can be used.

ML-IPsec has an unintended benefit in this case – it is actually more secure than the original IPsec, because the chosen plaintext attacks [Bel96] become more difficult now. Under the original IPsec, chosen plaintext attacks can compromise user data carried in an IPsec packet by exploiting the fact that

the values in many protocol header fields are predictable; that is, the encryption key can be discovered by comparing the plaintext and ciphertext versions of these fields. However, even so this will not compromise the user data under ML-IPsec, because it uses a different key from the headers.

4.3 AH and ESP Headers

The same security protocol formats, AH and ESP, are used in ML-IPsec. Both AH and ESP have transport mode or tunnel mode, as indicated by the "protocol mode" field of the designated SA. Figure 6 describes the format for both headers used in ML-IPsec.

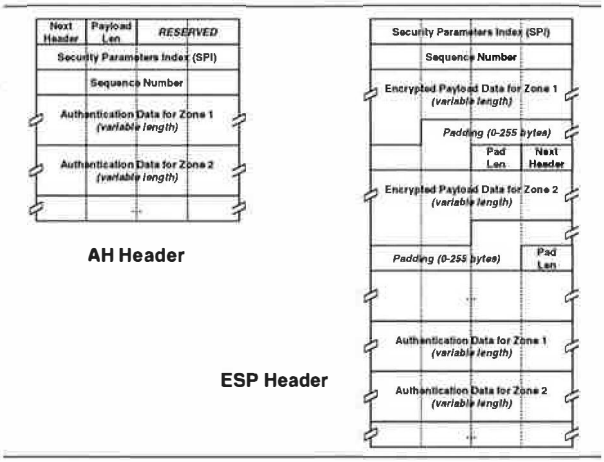


Figure 6: ML-IPsec Protocol Header Format

The protocol header format for AH in ML-IPsec is almost identical to the original IPsec AH [KA98a], except that the Authentication Data section in AH is further subdivided into zones. The Authentication Data field is a variable-length field that contains several Integrity Check Values (ICVs) for this packet. The total length of this field is controlled by Payload Len. The size of each ICV is determined by the authentication algorithm used in each zonal SA, but must be an integral multiple of 32 bits. The boundaries of these zonal authentication data sections can be derived from the CSA.

ML-IPsec is perhaps more useful in ESP, where the IP datagram can be encrypted using different keys in different SAs. The ML-IPsec ESP header format follows the principle in IPsec ESP. But unlike IPsec ESP, the Payload Data field in ML-IPsec ESP is broken into pieces, one for each zone. The Payload Data for each zone, together with Padding, Padding Length, and Next Header field (only in the designated zone), are collectively referred to as the ciphertext block for the zone. The size of each ciphertext block can be determined by the CSA, since all zones except the last one are fixed in size.

Similar to ML-IPsec AH, the optional Authentication Data field in ESP is also variable in length and contains several Integrity Check Values (ICVs) for this packet. The size of each ICV is determined by the authentication algorithm used in each zonal SA, but must be an integral multiple of 32 bits. The boundaries of these zonal authentication data sections can be derived from the CSA.

4.4 Inbound and Outbound Processing in ML-IPsec

4.4.1 ICV Calculation and Verification

The AH ICV calculation in ML-IPsec is rather different from that in the original IPsec. For the designated zone, the ICV is computed over:

- IP header fields that are immutable in transit.
- The AH header, including Next Header, Payload Len, Reserved, SPI, Sequence Number, and the Authentication Data (which is set to zero for this computation), and the optional explicit padding bytes if any.
- All octets in the designated zone.

For other non-designated zones, the ICV is computed only over the octets of the zone.

The ICV verification during the inbound processing of an ML-IPsec datagram is also done zone-by-zone. A zone is authenticated only if the corresponding zonal SA is non-null. The ICVs are calculated in the same way as described above, and the values are then matched against the ICVs stored in the Authentication Data. In an intermediate node, a packet will go through inbound processing and then outbound processing. If changes are made to the packet in an authorized zone, the ICV is recomputed and stored in a proper place in the Authentication Data field. The ICVs of the unchanged zones are left untouched.

4.4.2 Zone-by-Zone Encryption

On outbound processing, the sender takes the following steps in packet encryption:

1. **Zone-wise Encapsulation.** For each zone, all octets of all sub-zones are concatenated (in the order they appear in a datagram) and then encapsulated into the ESP Payload Data field for the corresponding zone.
2. **Padding.** The sender adds any necessary padding to each zone's Payload Data field, to meet the encryption algorithm's block size requirement if any, and to align it on a 4-byte boundary according to the ML-IPsec ESP format.
3. **Encryption.** The sender then encrypts the resulting plaintext (Payload Data, Padding, Pad Length, and Next Header) using the key, the encryption algorithm, and the algorithm mode indicated by the zonal SA and cryptographic synchronization data (if any).

4.4.3 Outbound processing

The outbound processing of an IP datagram in an IPsec gateway is illustrated in Figure 7 through a 2-zone example. The plaintext from each zone is masked by the zone map, concatenated into a continuous block, and passed through the zone-by-zone encryption as described above in Section 4.4.2. If a zone is itself a continuous block before the masking, we must do optimization to avoid extra copying; the encryption should be operated directly at the IP datagram buffer. For ESP, the ciphertexts from all zones are concatenated and stored in the ESP payload data field of an ML-IPsec ESP packet. After packet encryption, for AH and ESP with an authentication option, the sender computes the ICV from the ciphertext of each zone according to Section 4.4.1. The ICVs are then concatenated and stored in the Authentication Data field of the final outgoing ML-IPsec AH or ESP packet.

4.4.4 Inbound Processing

The inbound processing of an IPsec packet is almost a simple reverse of the outbound processing (see the same 2-zone example in Figure 8). If the SA structure indicated by the SPI value is a CSA type, the ML-IPsec processing mode is triggered. ML-IPsec first performs ICV verification if the protocol is AH, or if the protocol is ESP with an authentication option. The ICV check is done zone-by-zone according to Section 4.4.1. If the ICV check fails for any one zone, the entire datagram is discarded. The next step is the zone-by-zone decryption if the protocol is ESP. For a zone whose zonal SA is valid and non-null, the receiver decrypts the ESP Payload Data, Padding, Pad Length, and optional Next Header using the key, encryption algorithm, algorithm mode, and cryptographic synchronization data (if any), indicated by the zonal SA. After processing Padding, the receiver then reconstructs the original IP datagram from the original IP header (transport mode) or the tunnel IP header (tunnel mode), plus the IP payload stored in all the Payload fields. In the reverse procedure of encryption, the receiver takes Payload Data of a zone and restores the bytes back according to the zone map. If a zone has a null SA, the bytes corresponding to the zone map will be left zero.

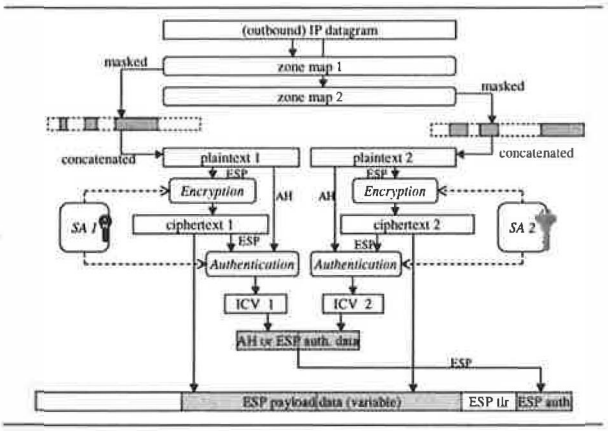


Figure 7: An Example of ML-IPsec Outbound Processing

4.4.5 Partial Datagram Processing at Intermediate Routers

In an intermediate node that is authorized to access at least one zone, a bypassing ML-IPsec datagram will go through inbound processing and then outbound processing if changes have been made (see Figure 9). The processing requires extra care because it may not have all the SAs to process the entire datagram. It must ignore zones for which

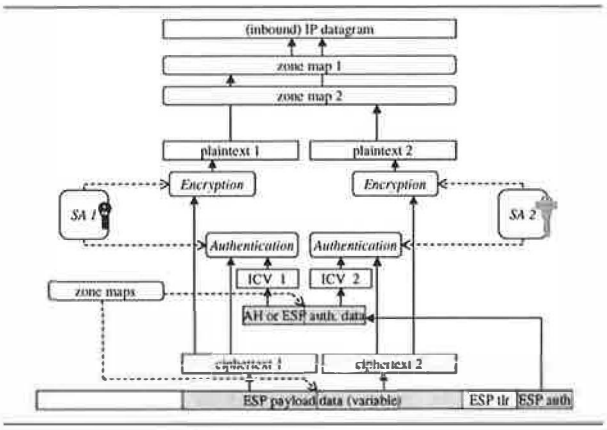


Figure 8: An Example of ML-IPsec Inbound Processing

keys are not granted. If the protocol is ESP, the decrypted plaintext may be incomplete for the original datagram, but it is likely to have the zone it needs (e.g., TCP header) for customized operations. If the intermediate node modifies the plaintext (e.g., in TCPPEP), it must redo authentication and/or encryption for that zone, and replace the corresponding ICV and/or Payload Data field in the bypassing IPsec datagram, before forwarding to the next hop.

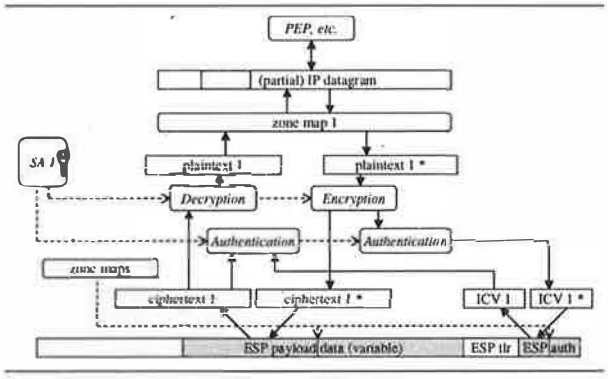


Figure 9: An Example of Partial Inbound/Outbound Processing

4.5 Bandwidth Overhead Analysis

The extra overhead introduced by the multi-layer protection model includes IPsec datagram size and processing load. The datagram size in a two-zone ML-IPsec is likely to increase when we do authentication or encryption as two separate plaintext blocks instead of one trunk in the original IPsec. For example, the concatenated ciphertext from two individually encrypted plaintexts might be larger than the single ciphertext of the concatenated plaintext, due to the synchronization data (such as an initialization vector in some encryption algorithm) and separate padding. The authentication data field is also bigger. For example, if HMAC-MD5-96 is used, the ICV is a fixed size of 12 bytes. If the CSA has two zones, the new IPsec datagram will increase by 12 bytes compared with the original IPsec one. To understand the increase in packet size caused by ML-IPsec, we conduct protocol analysis on TCP applications.

The datagram used in the analysis is a TCP datagram, with 20-byte IP header, 20-byte TCP header, no IP options, and no TCP options. For ML-IPsec, we assume the TCP datagram is divided into two zones, one for TCP/IP headers, and the other for the TCP payload. We calculate the overhead for both AH and ESP protocols (with authentication) and for both transport and tunnel mode. We analyze both ML-IPsec and IPsec for comparison purposes. In all the cases we assume use of the HMAC-MD5-96 algorithm for authentication and the 3DES-CBC algorithm for encryption.

We have analyzed the overhead for all eight cases (AH vs. ESP, Transport mode vs. Tunnel mode, and IPsec vs. ML-IPsec). Due to space limitations, we will not enumerate the calculation in detail. We only summarize the results in Tables 1 and 2. The variable n denotes the length of the TCP payload in the original IP datagram.

While the use of IPsec to protect IP datagrams adds an overhead ranging from 24 to 57 bytes, the new ML-IPsec scheme adds only an additional 12 bytes to the AH protocol and a maximum 20 bytes to the ESP protocol. Assuming an average IP datagram of 536 bytes, that is only a 2-3% increase. One way to further reduce the overhead increase is to use a "weaker" authentication algorithm for the "less important" field. For example, a 4-byte HMAC-MD5-32 ICV may be sufficient for the TCP header zone, instead of the 12-byte HMAC-MD5-96 ICV in the original IPsec. This saves 8 bytes for each ML-IPsec packet and brings the overhead down to the 1-2% range.

Table 1: Packet Length Comparison (bytes)

	Original IP	IPsec	ML-IPsec
AH Transport mode	$40+n$	$64+n$	$76+n$
AH Tunnel mode	$40+n$	$84+n$	$96+n$
ESP Transport mode	$40+n$	$64+\lceil(6+n)/8\rceil*8$	$92+\lceil(1+n)/8\rceil*8$
ESP Tunnel mode	$40+n$	$88+\lceil(2+n)/8\rceil*8$	$116+\lceil(1+n)/8\rceil*8$

Table 2: Packet Length Overhead (bytes)

	IP → IPsec	IP → ML-IPsec	IPsec → ML-IPsec
AH Transport mode	24	36	12
AH Tunnel mode	44	56	12
ESP Transport mode	[30,37]	[46,53]	12 or 20
ESP Tunnel mode	[50,57]	[70,77]	20

5 Implementation

5.1 Platform

This implementation of ML-IPsec is done on Linux FreeS/WAN version 1.1 on Linux kernel version 2.2.12. Linux FreeS/WAN (`www.freeswan.org`) is an implementation of IPsec available free (under GNU license term) to users and developers all around the world. The FreeS/WAN system has the following major parts.

- **KLIPS.** The **K**ernel **IP**sec **S**upport part includes the necessary elements in the Linux kernel for running IPsec protocols on the system. Most of the changes required for implementing ML-IPsec using FreeS/WAN are done in this part.
- **The Pluto Daemon.** This part implements the IKE protocol [HC98] – verifying identities, choosing security policies and negotiating keys for the KLIPS layer. Our current ML-IPsec implementation does not change this part, as we are using manual keying only. Our future work will involve modifying the Pluto daemon to facilitate automatic keying among all nodes and specifying ML-IPsec policies.
- **The `ipsec` Command.** This is a user command for controlling IPsec activities, such as setting up and tearing down IPsec tunnels, etc.
- **Linux FreeS/WAN Configuration File.** This file (usually `/etc/ipsec.conf`) contains configuration parameters for setting up IPsec tunnels in the system. We have modified the format of this file to accommodate ML-IPsec functions. The modifications will be listed in a later section.

5.2 Changes in Data Structure

The main changes that we have made are in the data structures for SAs (Security Associations). In original FreeS/WAN, the SAD (security association database) was implemented as a hash table of `struct tdb` nodes. Our ML-IPsec implementation does not change this storage organization, but it modifies the `tdb` structure to store extra fields for ML-IPsec specific data. In addition, we create new structures for "zonemap" and "subzone" as illustrated in the design. The case of a "normal" SA (relating to the the original IPsec) would be handled as a special case of an ML-IPsec SA with one zonemap extending from byte 1 to EOP (end-of-packet). Figure 10 contains a verbatim copy of the new data structure; our changes are annotated by the C preprocessor macro `ML_IPSEC`. The storage organization of these structures in the memory is shown in Figure 12.

Although the AH and ESP header formats in ML-IPsec are different from the original IPsec, the correpsonding programming constructs do not need modification because the fixed length fields remain the same. The variable length fields and the pointer targets are set up properly during the allocation stage for the socket buffers.

The data structure of the control interface (`encap_msghdr`) used for passing the requisite infor-

```
struct tdb                      /* tunnel descriptor block */
{

#ifdef ML_IPSEC
  struct zone         *csa_zonemap;                 /* pointer to the zonemap for the CSA */
  __u8                 desig_sa_flag;               /* boolean value - designated SA or not*/
#endif

  struct tdb          *tdb_hnext;                   /* next in hash chain */
  struct tdb          *tdb_onext;                   /* next in output */
  struct tdb          *tdb_inext;                   /* next in input (prev!) */
  struct ifnet        *tdb_rcvif;                   /* related rcv encap interface */
  struct sa_id         tdb_said;                    /* SA ID */
  __u32                tdb_seq;                     /* seq num of msg that set this SA */
  __u32                tdb_pid;                     /* PID of process that set this SA */
  struct xformsw      *tdb_xform;                   /* transformation to use (host order)*/
  caddr_t              tdb_xdata;                   /* transformation data (opaque) */
  __u8                 tdb_authalg;                 /* auth algorithm for this SA */
  __u8                 tdb_encalg;                  /* enc algorithm for this SA */

  __u8                 tdb_replaywin;               /* replay window size */
  __u8                 tdb_state;                   /* state of SA */
  __u32                tdb_replaywin_lastseq;       /* last pkt sequence num */
  __u64                tdb_replaywin_bitmap;        /* bitmap of received pkts */

  __u32                tdb_flags;                   /* generic xform flags */

  __u32                tdb_lifetime_allocations_c;  /* see rfc2367 */
  __u32                tdb_lifetime_allocations_s;
  __u32                tdb_lifetime_allocations_h;
  __u64                tdb_lifetime_bytes_c;
  __u64                tdb_lifetime_bytes_s;
  __u64                tdb_lifetime_bytes_h;
  __u64                tdb_lifetime_addtime_c;
  __u64                tdb_lifetime_addtime_s;
  __u64                tdb_lifetime_addtime_h;
  __u64                tdb_lifetime_usetime_c;
  __u64                tdb_lifetime_usetime_s;
  __u64                tdb_lifetime_usetime_h;
  struct sockaddr     *tdb_addr_s;                  /* src sockaddr */
  struct sockaddr     *tdb_addr_d;                  /* dst sockaddr */
  struct sockaddr     *tdb_addr_p;                  /* proxy sockaddr */
  __u16                tdb_addr_s_size;
  __u16                tdb_addr_d_size;
  __u16                tdb_addr_p_size;
  __u16                tdb_key_bits_a;              /* size of authkey in bits */
  __u16                tdb_auth_bits;               /* size of authenticator in bits */
  __u16                tdb_key_bits_e;              /* size of enckey in bits */
  __u16                tdb_iv_bits;                 /* size of IV in bits */

  __u8                 tdb_iv_size;
  __u16                tdb_key_a_size;
  __u16                tdb_key_e_size;
  caddr_t              tdb_key_a;                   /* authentication key */
  caddr_t              tdb_key_e;                   /* encryption key */
  caddr_t              tdb_iv;                      /* Initialisation Vector */
  __u16                tdb_ident_type_s;            /* src identity type */
  __u16                tdb_ident_type_d;            /* dst identity type */
  __u64                tdb_ident_id_s;              /* src identity id */
  __u64                tdb_ident_id_d;              /* dst identity id */
  __u8                 tdb_ident_len_s;             /* src identity type */
  __u8                 tdb_ident_len_d;             /* dst identity type */
  caddr_t              tdb_ident_data_s;            /* src identity data */
  caddr_t              tdb_ident_data_d;            /* dst identity data */
};


#ifdef ML_IPSEC
struct sub_zone
{
  struct sub_zone     *next;                        /* next subzone pointer for the zone*/
  __u16 left;                                       /* left bound of the subzone*/
  __u16 right;                                      /* right bound of the subzone*/
};

struct zone
{
  struct sub_zone     *subzone;                     /* pointer to the first subzone for this zone*/
  struct tdb          *corres_sa;                   /* pointer to the corresponding SA for this zone */
  struct zone         *next;                        /* next zone pointer for the CSA*/
};
#endif
```

Figure 10: The New SA Data Structure in FreeS/WAN Code for ML-IPsec

mation read from the configuration file to the kernel storage of SAs is also changed to accomodate the zone boundaries in the `Xfm` sub-structure (see Figure 11).

```
struct
{
  struct sa_id Said;      /* SA ID */
  int If;                 /* enc i/f for input */
  int Alg;                /* Algorithm to use */

#ifdef ML_IPSEC           /* zone bytes specifications */
  __u16 sbyte[MAX_SUBZONES];
  __u16 ebyte[MAX_SUBZONES];
#endif

  union {                 /* Data */
    __u8 Dat[1];
    __u64 Datq[1];        /* maximal alignment (?) */
  } u;
} Xfm;
```

Figure 11: The New `Xfm` Structure

5.3 Changes in C Functions

A number of C functions in the FreeS/WAN system source files need to be altered to take care of the changes made to the `tdb` data structure. Most of these files belong to the KLIPS part of the FreeS/WAN system. Rest are the user level files that interact with the KLIPS through the device interface.

The functions like `deltdb()`, `deltdbchain()` and `ipsec_tdbcleanup()`, which deal with deletion of SAs from the SAD, require minor changes to take care of extra pointers for zones and subzones in the CSAs. Some other functions requiring minor changes are `ipsec_spi_get_info()` and `ipsec_spigrp_get_info()`, which are responsible for printing information in the `/proc` directory. Some extra information relating to CSAs (like zone lengths and subzones) needs to be displayed in this case.

The major changes are in the functions that deal with user level process interaction with the kernel in storing, extracting, and updating SAs in the SAD. `ipsec_callback()` deals with the interaction of the user level `ipsec` command with the ipsec device and takes care of adding, deleting, and updating SAs from the SAD in the kernel. It calls the functions mentioned above and also functions like `tdb_init()`, which facilitate updating the fields in the kernel SA using the information received from the user level process through the device interface. A major change in this function is the formation of the SAD structure as shown in Figure 12, which now includes chaining between CSA and Zonal SAs using zonemaps and subzones.

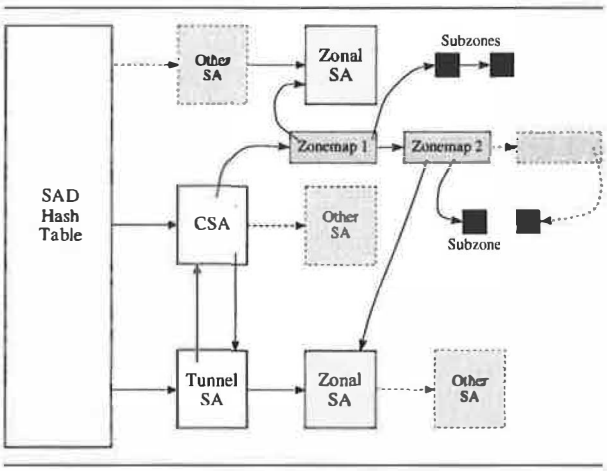


Figure 12: ML-IPsec storage structure in the SAD

The inbound and outbound processing of FreeS/WAN also requires major changes for ML-IPsec. The function `ipsec_rcv()` does the inbound processing for an IPsec packet received over the network. This function is responsible for getting the packet from the socket buffers, decapsulating it, extracting information from it to locate the SA in the kernel, doing authentication if necessary, doing decryption if necessary and forming the original IP Packet. It then pushes the packet up the stack to the IP layer for processing and/or forwarding onto the internal network. This function requires considerable changes to facilitate the processing in accordance with the new `tdb` structure and the new AH and ESP headers. The case of partial authentication/decryption (new in ML-IPsec, see Section 4.4.5) is also handled in this function.

The function `ipsec_tunnel_start_xmit()` is responsible for the outbound IPsec processing of an IP packet. The activities carried out by this function include: receiving packets from the IP layer, finding the corresponding SA and encapsulating route, tunneling the packet if necessary, and attaching the corresponding ESP and AH headers if necessary. The changes made to this function include differential calculation of the extra space required for putting zonal headers and trailers, buffer management on a per zone basis and authentication and encryption on a per zone basis. Compared to IPsec, this function has a much more complicated task because, instead of encapsulating and processing (encryption

and/or authentication) on the whole payload, it has to take care of zone boundaries within the payload and provide differential processing according to the particular zonal SA. Futhermore, the encapsulation has to be exactly the same as in the IPsec case.

Finally, a number of changes are made in the shell scripts that interpret the configuration file and call the `ipsec` command with appropriate arguments for controlling the ML-IPsec functionality.

So far, we have omitted the PF_KEY interface [MMP98] part, which also operate on the `tdb` structures to facilitate communication between the user level utilities and the kernel level processes. The functions in this interface will ultimately need appropriate changes as part of our future work to add automatic keying support.

5.4 Changes in Configuration File

The `ipsec.conf` file specifies most configuration and control information for the FreeS/WAN IPsec subsystem. Its syntax needs to be modified for use in ML-IPsec implementation. This includes adding a new section called the ZONE section, and adding specification for six new parameters. Further, some parameters ought to be moved around from one section to another to match new SA structure. Most of these changes apply to the CONN section. The CONFIG section needs no change. Here we explain the modification in detail.

CONN Sections Two sets of new parameters have been added to this section and seven parameters have been moved to the new ZONE section. The semantics for the remaining parameters in this section stays the same as in original FreeS/WAN. The parameters that remain in this section are `type`, `auto`, `left`, `leftsubnet`, `leftnexthop`, `leftfirewall`, the corresponding "right" parameters, and the manual keying parameters – `spibase`, `espreplay_window`, and `ahreplay_window`. Other parameters that correspond to the automatic keying policy also remain unchanged, because the current ML-IPsec implementation only deals with manual keying.

The new parameters that we have added to this section are:

- *hop_number* This is an integer value which represents the number of intermediate nodes that will be doing ML-IPsec processing on the packets. This number does not include the source and the destination nodes.

- *hop[x]* This is the IP address of the intermediate node that will be doing ML-IPsec processing on the packet from left to right and vice-versa. There can be multiple instances of this parameter on separate lines in the file as long as x satisfies the condition $1 \leq x \leq n$ (where n is the value of the `hop_number` parameter) and is not reused. The value of each `hop[`x`]` parameter is also not to be reused. Also if `hop[`x`]` is present then `hop[`$x-1$`]` should also be present. The syntax of this value is the same as a `left` parameter value.

- *zones* This is a comma-separated list of `ZONE` section names that belong to this connection.

ZONE Sections This new section is designed to cohere with the ML-IPsec zone concept. The parameters that are moved from the CONN sections in original FreeS/WAN include `esp`, `espenckey`, `espauthkey`, `leftespspi`, `ah`, `ahkey` and `leftahspi`. The semantics and syntax of these parameters remain the same as before. The new parameters are:

- *hosts* This is a comma-separated list of the string "*hop[x]*" where x is as defined in the previous section. This defines the list of "hops" that have access to this zone of the IP packet. The IP addresses for the "hops" are defined in the above CONN section, but the list of "hops" that can access this zone are defined here. For example,

  hop[2], hop[4], hop[5]

  means that the nodes whose IP addresses are defined by hop[2], hop[4] and hop[5] in the conn section have access to this zone of the packet.

- *zonebytes* This is a comma-separated list of integer ranges. It specifies the bytes of an IP packet that constitute a zone, like

  1-14, 20-26, 42-EOP

  (EOP denotes end-of-packet).

- *desig* This yes/no value signifies whether this section specifies a designated zone or not.

6 Conclusion

The end-to-end network security mechanisms such as IPsec and the rich network services such as those described in this paper are two fundamentally conflicting mechanisms. On the one hand, end-to-end security advocates the use of cryptography at the network layer to protect the payload over an untrustworthy internet. On the other hand, certain network services rely on intermediate nodes to perform "intelligent operations" based on the packet data type – the information encoded in a higher protocol layer. It is the need to find the right balance between the two mechanisms and to achieve the goals of both, that makes for a difficult engineering problem.

Our attempt to solve the problem is based on the layering architecture for network security protocols. The approach presented in this paper may already have the right mix to provide both security and extensibility in one unified platform. Certainly, we have shown that through protocol design and system implementation ML-IPsec can easily be added to an existing IPsec system and that its overhead is low. ML-IPsec has achieved the goal of granting trusted intermediate routers a secure, controlled, and limited access to selected portions of IP datagrams, while preserving the end-to-end security protection to user data. A similar system, which is based on the same layering principle described in this paper, has been implemented independently by University of Maryland [Kar99], although their implementation was based on an older version of FreeS/WAN and an older version of Linux kernel.

Our plan for future work includes an extension of IKE to support ML-IPsec. IKE is the key distribution protocol for IPsec, but we did not use it here because our current implementation uses manual keying only. It will be very important for ML-IPsec to be able to utilize automatic keying because it uses more keys, involves intermediate nodes, and requires a more complicated configuration than the original IPsec. The technical challenge will be finding the efficient mechanism needed for multi-party key distribution.

Web Site

The URL for the web site of this project is `http://www.wins.hrl.com/people/ygz/ml-ipsec` and it contains relevant documents, software releases (when they are ready), performance measure data for this implementation, and further developments for ML-IPsec.

Acknowledgments

The authors would like to thank the staff and students in the WINS group at HRL for many good suggestions and criticisms. In particular, we'd like to acknowledge the following students who have interned with us. Ilan Zohar (`ilanz@leland.stanford.edu`) of Stanford University, started looking into this problem in summer 1998. Praveen Kumar Gonugunta (`gonugunt@uiuc.edu`), a Ph.D. student at UIUC, started the first prototype implementation of ML-IPsec on an early version of FreeS/WAN (1.0, on Linux kernel 2.0.36), when he was an intern with us in summer 1999. Ashish Shah (`ashah@leland.stanford.edu`), a Ph.D. student at Stanford University who also interned with us in 1999, helped in the first prototype. In addition, the authors have also benefited from their e-mail discussions with participants in the IETF IPSEC working group and TF-ESP BOF.

References

[Bel96] S. Bellovin. Problem areas for the IP security protocols. In *Proceedings of the Sixth Usenix Unix Security Symposium*, San Jose, CA, July 1996.

[Bel99] S. Bellovin. Transport-friendly ESP (or layer violation for fun and profit). IETF-44 TF-ESP BOF, March 1999.

[BKGM00] J. Border, M. Kojo, J. Griner, and G. Montenegro. Performance enhancing proxies, March 2000. IETF draft, work in progress `draft-ietf-pilc-pep-02.txt`.

[BPSK97] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A comparison of mechanism for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, December 1997.

[DA99] T. Dierks and C. Allen. The TLS protocol, version 1.0, January 1999. IETF RFC 2246.

[FF99] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking*, August 1999.

[FJ93] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, pages 397–413, August 1993.

[HC98] D. Harkins and D. Carrel. The Internet key exchange (IKE), November 1998. IETF RFC 2409.

[KA98a] S. Kent and R. Atkinson. IP authentication header, November 1998. IETF RFC 2402.

[KA98b] S. Kent and R. Atkinson. IP encapsulating security payload (ESP), November 1998. IETF RFC 2406.

[KA98c] S. Kent and R. Atkinson. Security architecture for the Internet protocol, November 1998. IETF RFC 2401.

[Kar99] M. Karir. IPSEC and the Internet, December 1999. Master Thesis, University of Maryland (Technical Report ISR-MS-99-14).

[MMP98] D. McDonald, C. Metz, and B. Phan. PF_KEY key management API, version 2, July 1998. IETF RFC 2367.

[NBB99] D. Nessett, B. Braden, and S. Bellovin. IPSEC: Friend or Foe. Panel discussion in Network and Distributed System Security Symposium (NDSS'99), February 1999.

[ZDRD97] Y. Zhang, D. DeLucia, B. Ryu, and S. Dao. Satellite communications in the global Internet: Issues, pitfalls, and potential. In *INET'97*, Kuala Lumpur, Malaysia, June 1997.

Defeating TCP/IP Stack Fingerprinting

Matthew Smart G. Robert Malan Farnam Jahanian
Department of Electrical Engineering and Computer Science
University of Michigan
1301 Beal Ave.
Ann Arbor, Mich. 48109-2122
{mcsmart,rmalan,farnam}@eecs.umich.edu

Abstract

This paper describes the design and implementation of a TCP/IP stack fingerprint scrubber. The fingerprint scrubber is a new tool to restrict a remote user's ability to determine the operating system of another host on the network. Allowing entire subnetworks to be remotely scanned and characterized opens up security vulnerabilities. Specifically, operating system exploits can be efficiently run against a pre-scanned network because exploits will usually only work against a specific operating system or software running on that platform. The fingerprint scrubber works at both the network and transport layers to convert ambiguous traffic from a heterogeneous group of hosts into sanitized packets that do not reveal clues about the hosts' operating systems. This paper evaluates the performance of a fingerprint scrubber implemented in the FreeBSD kernel and looks at the limitations of this approach.

1 Description

TCP/IP stack fingerprinting is the process of determining the identity of a remote host's operating system by analyzing packets from that host. Freely available tools (such as `nmap` [3] and `queso` [15]) exist to scan TCP/IP stacks efficiently by quickly matching query results against a database of known operating systems. The reason this is called "fingerprinting" is therefore obvious; this process is similar to identifying an unknown person by taking his or her unique fingerprints and finding a match in a database of known fingerprints. The difference is that in real fingerprinting, law enforcement agencies use fingerprinting to track down suspected criminals; in computer networking potential attackers can use fingerprinting to quickly create a list of targets.

We argue that fingerprinting tools can be used to aid unscrupulous users in their attempts to break into or disrupt computer systems. A user can build up a profile of IP addresses and corresponding operating systems for later attacks. `Nmap` can scan a subnetwork of 254 hosts in only a few seconds, or it can be set up to scan very slowly, i.e. over days. These reports can be compiled over weeks or months and cover large portions of a network. When someone discovers a new exploit for a specific operating system, it is simple for an attacker to generate a script to run the exploit against each corresponding host matching that operating system. An example might be an exploit that installs code on a machine to take part in a distributed denial of service attack. Fingerprinting scans can also potentially use non-trivial amounts of network resources including bandwidth and processing time by intrusion detection systems and routers.

Fingerprinting provides fine-grained determination of an operating system. For example, `nmap` has knowledge of 21 different versions of Linux. Other methods of determining an operating system are generally coarse-grained because they use application-level methods. An example is the banner message a user receives when he or she uses telnet to connect to a machine. Many systems freely advertise their operating system in this way. This paper does not deal with blocking application-level fingerprinting because it must be dealt with on an application by application basis.

Almost every system connected to the Internet is vulnerable to fingerprinting. The major operating systems are not the only TCP/IP stacks identified by fingerprinting tools. Routers, switches, hubs,

bridges, embedded systems, printers, firewalls, web cameras, and even game consoles are identifiable. Many of these systems, like routers, are important parts of the Internet infrastructure, and compromising infrastructure is a more serious problem than compromising end hosts. Therefore a general mechanism to protect any system is needed.

Some people may consider stack fingerprinting a nuisance rather than a security attack. As with most tools, fingerprinting has both good and bad uses. Network administrators should be able to fingerprint machines under their control to find known vulnerabilities. Stack fingerprinting is not necessarily illegal or an indication of malicious behavior, but we believe the number of scans will grow in frequency as more people access the Internet and discover easy to use tools such as `nmap`. As such, network administrators may not be willing to spend time or money tracking down what they consider petty abuses each time they occur. Instead they may choose to reserve their resources for full-blown intrusions. Also, there may be networks that no single authority has administrative control over, such as a university residence hall. A tool that detects fingerprinting scans but turns them away would allow administrators to track attempts while keeping them from penetrating into local networks.

This paper presents the design and implementation of a tool to defeat TCP/IP stack fingerprinting. We call this new tool a *fingerprint scrubber*. The fingerprint scrubber is transparently interposed between the Internet and the network under protection. The intended use of the scrubber is for it to be placed in front of a set of end hosts or a set of network infrastructure components. The goal of the tool is to block the majority of stack fingerprinting techniques in a general, fast, scalable, and transparent manner.

We describe an experimental evaluation of the tool and show that our implementation blocks known fingerprint scan attempts and is prepared to block future scans. We also show that our fingerprint scrubber can match the performance of a plain IP forwarding gateway on the same hardware and is an order of magnitude more scalable than a transport-level firewall.

The remaining sections are organized as follows. We describe TCP/IP stack fingerprinting in more detail in Section 2. In Section 3 we describe the design and implementation of our fingerprint scrubber. In Section 4 we evaluate the validity and performance of the scrubber. In Section 5 we cover related work and in Section 6 we cover future directions. Finally, in Section 7 we summarize our work.

2 TCP/IP Stack Fingerprinting

The most complete and widely used TCP/IP fingerprinting tool today is `nmap`. It uses a database of over 450 fingerprints to match TCP/IP stacks to a specific operating system or hardware platform. This database includes commercial operating systems, routers, switches, firewalls, and many other systems. Any system that speaks TCP/IP is potentially in the database, which is updated frequently. `Nmap` is free to download and is easy to use. For these reasons, we are going to restrict our talk of existing fingerprinting tools to `nmap`.

`Nmap` fingerprints a system in three steps. First, it performs a port scan to find a set of open and closed TCP and UDP ports. Second, it generates specially formed packets, sends them to the remote host, and listens for responses. Third, it uses the results from the tests to find a matching entry in its database of fingerprints.

`Nmap` uses a set of nine tests to make its choice of operating system. A test consists of one or more packets and the responses received. Eight of `nmap`'s tests are targeted at the TCP layer and one is targeted at the UDP layer. The TCP tests are the most important because TCP has a lot of options and variability in implementations. `Nmap` looks at the order of TCP options, the pattern of initial sequence numbers, IP-level flags such as the don't fragment bit, the TCP flags such as RST, the advertised window size, and a few more things. For more details, including the specific options set in the test packets, refer to the home page for `nmap` [3].

Figure 1 is an example of the output of `nmap` when scanning our EECS department's web server, www.eecs.umich.edu, and one of our department's printers. The TCP sequence prediction result comes from `nmap`'s determination of how a host increments its initial sequence number for each TCP connection. Many commercial operating systems use a random, positive increment, but simpler systems tend to use fixed increments or increments based on the time between connection attempts.

(a)

```
TCP Sequence Prediction:
            Class=truly random
            Difficulty=9999999 (Good luck!)
Remote operating system guess:
            Linux 2.0.35-37
```

(b)

```
TCP Sequence Prediction:
            Class=trivial time dependency
            Difficulty=1 (Trivial joke)
Remote operating system guess:
            Xerox DocuPrint N40
```

Figure 1: Output of an nmap scan against (a) a web server running Linux and (b) a shared printer.

While nmap contains a lot of functionality and does a good job of performing fine-grained fingerprinting, it does not implement all of the techniques that could be used. Various timing-related scans could be performed. For example, determining whether a host implements TCP Tahoe or TCP Reno by imitating packet loss and watching recovery behavior. We discuss this threat and potential solutions in Section 3.2.4. Also, a persistent person could also use methods such as social engineering or application-level techniques to determine a host's operating system. Such techniques are outside the scope of this work. However, there will still be a need to block TCP/IP fingerprinting scans even if an application-level fingerprinting tool is developed. Currently, TCP/IP fingerprinting is the fastest and easiest method for identifying remote hosts' operating systems, and introducing techniques that target applications will not make it obsolete.

3 Fingerprint Scrubber

We developed a tool called a fingerprint scrubber to remove ambiguities from TCP/IP traffic that give clues to a host's operating system. In this section we discuss the goals and intended use of the scrubber and its design and implementation. We demonstrate the validity of the scrubber in the face of known fingerprinting scans and give performance results in the next section.

3.1 Goals and Intended Use of Fingerprint Scrubber

The goal of the fingerprint scrubber is to block known stack fingerprinting techniques in a general, fast, scalable, and transparent manner. The tool should be general enough to block classes of scans, not just specific scans by known fingerprinting tools. The scrubber must not introduce much latency and must be able to handle many concurrent TCP connections. Also, the fingerprint scrubber must not cause any noticeable performance or behavioral differences in end hosts. For example, it is desirable to have a minimal effect on TCP's congestion control mechanisms by not delaying or dropping packets unnecessarily.

We intend for the fingerprint scrubber to be placed in front of a set of systems with only one connection to a larger network. We expect that a fingerprint scrubber would be most appropriately implemented in a gateway machine from a LAN of heterogeneous systems (i.e. Windows, Solaris, MacOS, printers, switches) to a larger corporate or campus network. A logical place for such a system would be as part of an existing firewall. Another use would be to put a scrubber in front of the control connections of routers. The network under protection must be restricted to having one connection to the outside world because all packets traveling to and from a host must travel through the scrubber.

Because the scrubber affects only traffic moving through it, an administrator on the trusted side of the network will still be able to scan the network. Alternatively, an IP access list or some other authentication mechanism could be added to the fingerprint scrubber to allow authorized hosts to bypass scrubbing.

3.2 Fingerprint Scrubber Design and Implementation

We designed the fingerprint scrubber to be placed between a trusted network of heterogeneous systems and an untrusted connection (i.e. the Internet). The scrubber has two interfaces; one interface is designated as *trusted*, and the other is designated as *untrusted*. A packet coming from the untrusted interface is forwarded out the trusted interface and vice versa. The basic design principle is that data

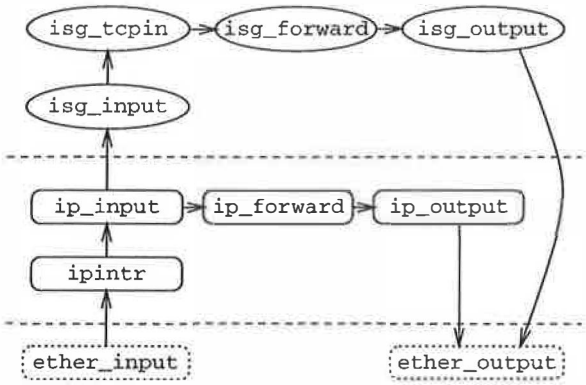


Figure 2: Data flow through modified FreeBSD kernel.

coming in from the untrusted interface is handled differently than data traveling out to the untrusted interface.

The fingerprint scrubber operates at the IP and TCP layers to cover a wide range of known and potential fingerprinting scans. We could have simply implemented a few of the techniques discussed in the following sections to defeat `nmap`. However, the goal of this work is to stay ahead of those developing fingerprinting tools. By making the scrubber operate at a generic level for both IP and TCP, we feel we have raised the bar sufficiently high.

The fingerprint scrubber is based off the protocol scrubber by Malan, et al. [7]. The protocol scrubber operates at the IP and TCP layers of the protocol stack. It is a set of kernel modifications to allow fast TCP flow reassembly to avoid TCP insertion and deletion attacks as described by Ptacek and Newsham [13]. The protocol scrubber follows TCP state transitions by maintaining a small amount of state for each connection, but it leaves the bulk of the TCP processing and state maintenance to the end hosts. This allows a tradeoff between the performance of a stateless solution with the control of a full transport-layer proxy. The protocol scrubber is implemented under FreeBSD, and we continued under FreeBSD 2.2.8 for our development.

Figure 2 shows the data flow through the kernel for the fingerprint scrubber. Packets come in from either the trusted or untrusted interface through an Ethernet driver. Incoming IP packets are handed to `ip_input` through a software interrupt, just as would be done normally. A filter in `ip_input` determines if the packet should be forwarded to the TCP scrubbing code. If not, then it follows the normal IP forwarding path to `ip_output`. If it is, then `isg_input` (ISG stands for Internet Scrubbing Gateway) performs IP fragment reassembly if necessary and passes the packet to `isg_tcpin`. Inside `isg_tcpin` the scrubber keeps track of the TCP connection's state. The packet is passed to `isg_forward` to perform TCP-level processing. Finally, `isg_output` modifies the next-hop link level address and `isg_output` or `ip_output` hands the packet straight to the correct device driver interface for the trusted or untrusted link.

We must also make sure that differences in the packets sent by the trusted hosts to the untrusted hosts don't reveal clues. These checks and modifications are done in `isg_forward` for TCP modifications, `isg_output` for IP modifications to TCP segments, and `ip_output` for IP modifications to non-TCP packets.

3.2.1 IP scrubbing

IP-level ambiguities arise mainly in IP header flags and fragment reassembly algorithms. Modifying flags requires no state but requires adjustment of the header checksum. Reassembly, however, requires fragments to be stored at the scrubber. Once a completed IP datagram is formed, it may need to be re-fragmented on the way out the interface.

The fingerprint scrubber uses the code in Figure 3 to normalize IP type-of-service and fragment bits in the header. This occurs for all ICMP, IGMP, UDP, TCP, and other packets for protocols built on top of IP. Uncommon and generally unused combinations of TOS bits are removed. In the case that these bits need to be used (i.e. an experimental modification to IP) this functionality could be removed. Most TCP/IP implementations we have tested ignore the reserved fragment bit and reset it to 0 if it is set, but we wanted to be safe so we mask it out explicitly. The don't fragment bit is reset if the MTU of the next link is large enough for the packet. This check is not shown in the figure.

Modifying the don't fragment bit could break MTU discovery through the scrubber. One could argue that the reason you would put the fingerprint scrubber in place is to hide information about the systems behind it. This might include topology and bandwidth information. However, such a modification is controversial. We leave the decision on whether or

```
/*
 * Normalize IP type-of-service flags
 */
switch (ip->ip_tos)
{
    case IPTOS_LOWDELAY:
    case IPTOS_THROUGHPUT:
    case IPTOS_RELIABILITY:
    case IPTOS_MINCOST:
    case IPTOS_LOWDELAY|IPTOS_THROUGHPUT:
        break;
    default:
        ip->ip_tos = 0;
}

/*
 * Mask out reserved fragment flag.
 * The MTU of the next downstream link
 * is large enough for the packet so
 * clear the don't fragment flag.
 */
ip->ip_off &= ~(IP_RF|IP_DF);
```

Figure 3: Code fragment to normalize IP header flags.

not to clear the don't fragment bit up to the end user by allowing the option to be turned off.

The fragment reassembly code is a slightly modified version of the standard implementation in the FreeBSD 2.2.8 kernel. It keeps fragments on a set of doubly linked lists. It first calculates a hash to determine which list the fragment maps to. A linear search is done over this list to find the IP datagram the fragment goes with and its place within the datagram. Old data in the fragment queue is always chosen over new data.

3.2.2 ICMP scrubbing

In this section we describe the modifications the fingerprint scrubber makes to ICMP messages. We only modify ICMP messages returning from the trusted side back to the untrusted side because fingerprinting relies on ICMP responses and not requests. Specifically, we modify ICMP error messages and rate limit all outgoing ICMP messages.

ICMP error messages are meant to include at least the IP header plus 8 bytes of data from the packet that caused the error. According to RFC 1812 [1], as many bytes as possible, up to a total ICMP packet length of 576 bytes, are allowed. However, nmap takes advantage of the fact that certain operating systems quote different amounts of data. To counter this we force all ICMP error messages coming from the trusted side to have data payloads of only 8 bytes by truncating larger data payloads. Alternatively, we could look inside of ICMP error messages to determine if IP tunneling is being used. If so, then we would allow more than 8 bytes.

3.2.3 TCP scrubbing

The TCP protocol scrubber we based the fingerprint scrubber on converts TCP streams into unambiguous flows by keeping a small amount of state per connection. The protocol scrubber keeps track of TCP connections using a simplified TCP state diagram. Basically, it keeps track of open connections by following the standard TCP three-way handshake (3WHS). This allows the fingerprint scrubber to block TCP scans that don't begin with a 3WHS. In fact, the first step in fingerprinting a system is typically to run a port scan to determine open and closed ports. Stealthy, meaning difficult to detect, techniques for port scanning don't perform a 3WHS and are therefore blocked. Only scans that commit to a 3WHS will get through.

A large amount of information can be gleaned from TCP options. We did not want to disallow certain options because some of them aid in the performance of TCP (i.e. SACK) yet are not widely deployed. Therefore we restricted our modifications to reordering the options within the TCP header. We simply provide a canonical ordering of the TCP options known to us. Unknown options are included after all known options. The handling of unknown options and ordering can be configured by the end user.

We also defeat attempts at predicting TCP sequence numbers by modifying the normal sequence number of new TCP connections. The fingerprint scrubber stores a random number when a new connection is initiated. Each TCP segment for the connection traveling from the trusted interface to the untrusted interface has its sequence number incremented by this value. Each segment for the connection traveling in the opposite direction has its acknowledgment number decremented by this value.

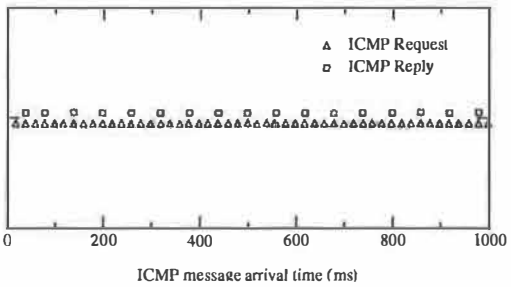


Figure 4: ICMP rate limiting of returning ICMP echo replies captured using `tcpdump`.

3.2.4 Timing attacks

The fingerprinting scans we have designed the fingerprint scrubber to block up to now have all been static, query-response style probes. A host carefully forms queries, sends them to a host, and analyzes the response or lack of response. Another possible form of scan is one that relies on timing responses. For example, the scanning host could open a TCP connection, simulate a packet loss, and watch the recovery behavior of the other host.

It would be very difficult to create a generic method for defeating timing-related scans, especially unknown scans. One approach would be to add a small, random amount of delay to packets going out the untrusted interface. The scrubber could even forward packets out-of-order. However this approach would introduce an increased amount of queuing delay and probably degrade performance. In addition, these measures are not guaranteed to block scans. For example, even with small amounts of random delay, it would be relatively easy to determine if a TCP stack implements TCP Tahoe or TCP Reno based on simulated losses because a packet retransmitted after an RTO has a much larger delay than one retransmitted because of fast retransmit.

We implemented protection against one possible timing-related scan. Some operating systems implement ICMP rate limiting, but they do so at different rates, and some don't do any rate limiting. We added a parameter for ICMP rate limiting to the fingerprint scrubber to defeat such a scan. The scrubber records a timestamp when an ICMP message travels from the trusted interface to the untrusted interface. The timestamps are kept in a small hash table referenced by the combination of the source and destination IP addresses. Before an ICMP message is forwarded to the outgoing, untrusted interface, it is checked against the cached timestamp. The packet is dropped if a certain amount of time has not passed since the previous ICMP message was sent to that destination from the source specified in the cache.

Figure 4 shows the fingerprint scrubber rate limiting ICMP echo requests and replies. In this instance, an untrusted host is sending ICMP echo requests once every 20 milliseconds using the `-f` flag with `ping` (flooding). The scrubber allows the requests through unmodified since we are not trying to hide the identity of the untrusted host from the trusted host. As the ICMP echo replies come back, however, the fingerprint scrubber makes sure that only those replies that come at least 50 ms apart are forwarded. Since the requests are coming 20 ms apart, for every three requests one reply will make it through the scrubber. Therefore the untrusted host receives a reply once every 60 ms.

We chose 50 ms for convenience because `ping -f` generates a stream of ICMP echo requests 20 ms apart, and we wanted the rate limiting to be noticeable. The exact value for a production system would have to be determined by an administrator or based upon previous ICMP flood attack thresholds. The goal was to homogenize the rate of ICMP traffic traveling from the untrusted interface to the trusted interface because operating systems rate limit their ICMP messages at different rates. Another method for confusing a fingerprinter would be to add small, random delays to each ICMP message. Such an approach would require keeping less state. We can add delay to ICMP replies, as opposed to TCP segments, because they won't affect network performance.

4 Evaluation of Fingerprint Scrubber

This section presents results from a set of experiments we performed to determine the validity, throughput, and scalability of the fingerprint scrubber. They show that our current implementation blocks known fingerprint scan attempts and can match the performance of a plain IP forwarding gateway on the same hardware. The experiments were conducted using a set of kernels with different fingerprint scrubbing options enabled for comparison.

The scrubber and end hosts each had 500 MHz Pen-

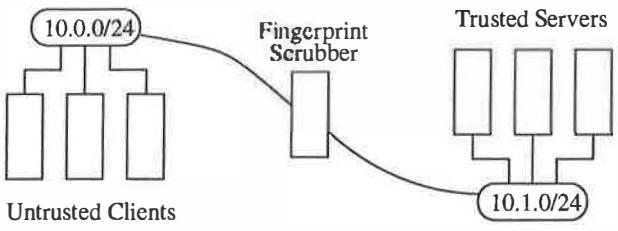


Figure 5: Experimental setup for measuring the performance of the fingerprint scrubber.

tium III CPUs and 256 megabytes of main memory. The end hosts each had one 3Com 3c905B Fast Etherlink XL 10/100BaseTX Ethernet card (`xl` device driver). The gateway had two Intel EtherExpress Pro 10/100B Ethernet cards (`fxp` device driver). The network was configured to have all traffic from 10.0.0/24 go to 10.1.0/24 through the gateway machine. Figure 5 shows how the three machines were connected as well as the trusted and untrusted domains.

4.1 Defeating fingerprint scans

To verify that our fingerprint scrubber did indeed defeat known scan attempts, we interposed our gateway between a set of machines running different operating systems. The operating systems we ran scans against under controlled conditions in our lab were FreeBSD 2.2.8, Solaris 2.7 x86, Windows NT 4.0 SP 3, and Linux 2.2.12. We also ran scans against a number of popular web sites, and campus workstations, servers, and printers.

`Nmap` was consistently able to determine all of the host operating systems without the fingerprint scrubber interposed. However, it was completely unable to make even a close guess with the fingerprint scrubber interposed. In fact, it wasn't able to distinguish much about the hosts at all. For example, without the scrubber `nmap` was able to accurately identify a FreeBSD 2.2.8 system in our lab. With the scrubber `nmap` guessed 14 different operating systems from three vendors. Each guess was wrong. Figure 6 shows a condensed result of the guesses `nmap` made against FreeBSD before and after interposing the scrubber.

The two main components that aid in blocking `nmap` are the enforcement of a three-way handshake for TCP and the reordering of TCP options. Many of `nmap`'s scans work by sending probes without the

(a)

```
Remote operating system guess:
   FreeBSD 2.2.1 - 3.2
```

(b)

```
Remote OS guesses:
   AIX 4.0 - 4.1, AIX 4.02.0001.0000,
   AIX 4.1, AIX 4.1.5.0, AIX 4.2,
   AIX 4.3.2.0 on an IBM RS/*,
   Raptor Firewall 6 on Solaris 2.6,
   Solaris 2.5, 2.5.1, Solaris 2.6 - 2.7,
   Solaris 2.6 - 2.7 X86,
   Solaris 2.6 - 2.7 with tcp_strong_iss=0,
   Solaris 2.6 - 2.7 with tcp_strong_iss=2,
   Sun Solaris 8 early acces beta (5.8)
     Beta_Refresh February 2000
```

Figure 6: (a) Operating system guess before fingerprint scrubbing and (b) after fingerprint scrubbing for an `nmap` scan against a machine running FreeBSD 2.2.8.

SYN flag set so they are discarded right away. Similarly, operating systems vary greatly in the order that they return TCP options. Therefore `nmap` suffers from a large loss in available information.

We intend this tool to be general enough to block potential or new scans also. We believe that the inclusion of IP header flag normalization and IP fragment reassembly aid in that goal even though we do not know of any existing tool that exploits such differences.

4.2 Throughput

We conducted an experiment to test the raw throughput possible through the fingerprint scrubber. The throughput was measured using the `netperf` benchmark [11]. The three test machines were connected using a 100 Mbps switch.

We measured both the throughput from the trusted side out to the untrusted side and from the untrusted side into the trusted side. This was to take into account our asymmetric filtering of the traffic. We ran experiments for TCP traffic to show the affect of a bulk TCP transfer and for UDP to exercise the fragment reassembly code. We used three kernels on the gateway machine to test different functionality of the fingerprint scrubber. The IP forwarding kernel is the unmodified FreeBSD

IP Forwarding	87.06
Fingerprint Scrubbing	86.86
Fingerprint Scrub. + Frag. Reas.	87.00
Application-level Transport Proxy	86.53

Table 1: Throughput for a single untrusted host to a trusted host using TCP (Mbps, ±2.5% at 99% CI).

IP Forwarding	87.06
Fingerprint Scrubbing	86.79
Fingerprint Scrub. + Frag. Reas.	86.84
Application-level Transport Proxy	86.53

Table 2: Throughput for a single trusted host to an untrusted host using TCP (Mbps, ±2.5% at 99% CI).

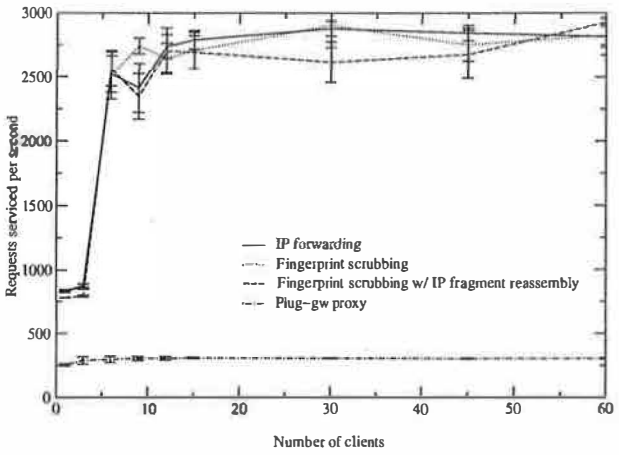


Figure 7: Connections per second through the gateway.

kernel, which we use as our baseline for comparison. The fingerprint scrubbing kernel includes the TCP options reordering, IP header flag normalization, ICMP modifications, and TCP sequence number modification but not IP fragment reassembly. The last kernel is the full fingerprint scrubber with fragment reassembly code turned on.

We also compared the fingerprint scrubber to a full application-level proxy. The TIS Firewall Toolkit's `plug-gw` proxy is an example of a firewall component that operates at the user-level to do transport-layer proxying [18]. When a new TCP connection is made to the proxy, `plug-gw` creates a second connection from the proxy to the server. The proxy's only job is to read and copy data from one connection to the other. A more fully featured firewall will process the copied headers and data, which adds additional latency and requires more state. Therefore the performance of `plug-gw` represents a minimum amount of work a firewall built from application-level proxies must perform. We modified the original `plug-gw` code so that it did no logging and no DNS resolutions, which resulted in a large performance increase. The proxy's kernel was also modified so that a large number of processes could be accommodated. A custom user-space proxy optimized for speed would certainly do better (the `plug-gw` proxy forks a child for each incoming connection). However, the multiple data copies and context switching will always resign any user-space implementation to significantly worse performance than in-kernel approaches [8; 17].

Table 1 shows the TCP bulk transfer results for an untrusted host connecting to a trusted host. Table 2 shows the results for a trusted host connecting to an untrusted host. The first result is that both directions show the same throughput. The second, and more important result, is that even when all of the fingerprint scrubber's functionality is enabled we are seeing a throughput almost exactly that of the plain IP forwarding. The bandwidth of the link is obviously the critical factor for all of the throughput experiments, therefore we would like to run these experiments again on a faster network in the future.

We ran the UDP experiment with the IP forwarding kernel and the fingerprint scrubbing kernel with IP fragment reassembly. Again, we measured both the untrusted to trusted direction and vice versa. To measure the affects of fragmentation, we ran the test at varying sizes up to the MTU of the Ethernet link and above. Note that 1472 bytes is the maximum UDP data payload that can be transmitted since the UDP plus IP headers add an additional 28 bytes to get up to the 1500 byte MTU of the link. The 2048 byte test corresponds to two fragments and the 8192 byte test corresponds to five fragments.

Table 3 shows the UDP transfer results for an untrusted host connecting to a trusted host. Table 4 shows the results for a trusted host connecting to an untrusted host. Once again both directions show the same throughput. We also see that the throughput of the fingerprint scrubber with IP fragment reassembly is almost exactly that of the plain IP forwarding. This is even true in the case of the 8192 byte test where the fragments must be reassembled

	64 bytes	1472 bytes	2048 bytes	8192 bytes
IP Forwarding	14.39	89.39	92.76	90.11
Fingerprint Scrubbing + Frag. Reas.	14.48	89.35	92.76	90.11

Table 3: Throughput for a single untrusted host to a trusted host using UDP (Mbps, ±2.5% at 99% CI).

	64 bytes	1472 bytes	2048 bytes	8192 bytes
IP Forwarding	14.39	89.39	92.76	90.11
Fingerprint Scrubbing + Frag. Reas.	14.40	89.37	92.76	90.12

Table 4: Throughput for a single trusted host to an untrusted host using UDP (Mbps, ±2.5% at 99% CI).

at the gateway and then re-fragmented before being sent out.

4.3 Scalability

We also ran an experiment to measure the scalability of the fingerprint scrubber. That is, how many concurrent TCP connections can our fingerprint scrubbing gateway support? We set up three machines as web servers to act as sinks for HTTP requests. On three other machines we ran increasing numbers of clients repeatedly requesting the same 1 KB file from the web servers. The choice of 1 KB allows us to keep the web servers' CPUs from being the limiting factor. Instead, the bandwidth of the link is again the bottleneck. The clients were connected with a 100 Mbps hub and the servers were connected with a 100 Mbps switch. The number of connections per second being made through the fingerprint scrubber was measured on the hub.

Figure 7 shows the number of sustained connections per second measured for plain IP forwarding, TCP/IP fingerprint scrubbing, fingerprint scrubbing with IP fragment reassembly, and the `plug-gw` application-level proxy. The error bars represent the standard deviation for each second. The results of the experiment are that the fingerprint scrubber scales comparably to the unmodified IP forwarder and performs much better than the transport proxy. The fingerprint scrubber achieves a rate of about 2,700 connections per second, which may be enough for most LANs. In comparison, the `plug-gw` proxy only achieves a rate of about 300 connections per second, which is an order of magnitude worse than the scrubber. The abysmal performance of the application-level proxy can be explained by the number of interrupts, data copies, and context switches incurred by such a user-level process. For each TCP connection, the proxy has to keep track of two complete TCP state machines and copy data up from the kernel then back down. The system running `plug-gw` was CPU bound for all but a few concurrent clients.

Achieving full line-speed in the fingerprint scrubber for higher bandwidth links would probably require dedicated hardware. A platform such as Intel's Internet Exchange Architecture (IXA) [14; 5] could help. The small size of the TCP state table we use in the fingerprint scrubber would be amenable to such a system.

5 Related Work

Current firewall technology is similar to the fingerprint scrubber [2]. Firewalls exist at the border of a network to provide access control. They both require packets to travel through them to get to their final destinations and can deny certain types of packets. Older firewalls, such as the TIS Firewall Toolkit [18], use application-level proxies and don't scale very well because they must keep two TCP connections open per session. Such a firewall will block TCP fingerprinting scans because it isolates the behavior of the TCP implementations on the side it is protecting by copying data. However, we showed that such a firewall's performance is an order of magnitude worse than the fingerprint scrubber. Also, the application-level proxy does not take care of IP-level ambiguities. Modern firewalls, such as Gauntlet [16], identify authorized flows by examining portions of packet headers and data payloads or using more sophisticated authentication meth-

ods. The firewall then routes packets through a fast path once the flow is set up to increase throughput and scalability. Such a firewall won't provide continued security against fingerprinting scans once a connection is set up. In contrast, the fingerprint scrubber removes scans throughout the lifetime of a flow while remaining more scalable by keeping a minimal amount of state per connection.

Various tools are available to secure a single machine against `nmap`'s operating system fingerprinting. The TCP/IP traffic logger `iplog` [10] can detect an `nmap` fingerprint scan and send out a packet designed to confuse the results. Other tools and operating system modifications simply use state inherently kept in the TCP implementation to drop certain scan types. However, none of these tools can be used to protect an entire network of heterogeneous systems. In addition, these methods will not work for networks that are not under single administrative control, unlike the fingerprint scrubber.

Vern Paxson presents a tool to analyze a TCP implementation's behavior called `tcpanaly` [12]. It works offline on `tcpdump` traces to try to distinguish if a certain traffic pattern is consistent with an implementation. In this way, it is doing a sort of TCP fingerprinting. However, `tcpanaly` suffers from a lot of uncertainty that makes it unfeasible as a fingerprinting tool. It also keeps explicit knowledge of several TCP/IP implementations. In contrast, our fingerprint scrubber has no knowledge of other implementations. The main contribution `tcpanaly` makes is not in fingerprinting but in analyzing the correctness of a TCP implementation and aiding in determining if an implementation has faults.

Malan, et al. [7] have presented the idea of not only transport-level scrubbing, but also application-level scrubbing. Obviously more specialization would need to be done. The main focus is on HTTP traffic to protect web servers. The idea could be extended to protect infrastructure components such as routers by scrubbing RIP, OSPF, and BGP.

6 Future Work

As mentioned in Sections 4.2 and 4.3, the first-order limiting factor in the performance of the fingerprint scrubber is the available link bandwidth. We are planning on testing the scrubber over gigabit Ethernet connections. To support a ten-fold increase in bandwidth we will be looking at reducing the amount of data copying, using incremental checksums when modifying header bits, and using a faster fragment reassembly algorithm.

Because of the close relationship between firewalls and fingerprint scrubbers, we would like to combine the two technologies. We would use the scrubber as a substrate and add features, such as authentication, required by a fully functional firewall. We believe such a system would combine the additional security benefits of a modern firewall with the performance characteristics and benefits of the fingerprint scrubber.

We would also like to examine how IP security [6] affects TCP/IP stack fingerprinting and operating system discovery. If a host implementing IPsec doesn't allow unknown hosts to establish connections, then those hosts will not be able to discern the host's operating system because all packets will be dropped. If a host does allow unknown hosts to connect in tunnel mode, however, then a fingerprint scrubber will be ineffective. The scrubber will be unable to examine and modify the encrypted and encapsulated IP and TCP headers. However, allowing any host to make a secure connection to an IPsec-enabled host is not the standard procedure unless it is a public server. Another portion of IPsec that could be exploited is the key exchange protocols, such as ISAKMP/IKE [9; 4]. If different systems have slight differences in their implementations, a scanner might be able to discern the host's operating system.

Another thing we would like to try is to have the fingerprint scrubber spoof an operating system's fingerprint instead of anonymizing it. For example, it might be interesting to have all of the computers on your network appear to be running the secure operating system OpenBSD. This is harder to do than simply removing ambiguities because you have to introduce artifacts in enough places to make the deception plausible.

As network infrastructure components increase in speed, tools such as the fingerprint scrubber must scale to meet the demand. To try to achieve line-speed, we would like to implement core components of the fingerprint scrubber in hardware. An example would be to build the minimal TCP state machine we use into a platform such as Intel's Internet Exchange Architecture (IXA) [14; 5].

7 Conclusions

We presented a new tool called a fingerprint scrubber to remove clues about the identity of an end host's operating system. We showed that the scrubber blocks known fingerprinting scans, is comparable in performance to a plain IP forwarding gateway, and is significantly more scalable than a full transport-layer firewall.

The fingerprint scrubber successfully and completely blocks known scans by removing many clues from the IP and TCP layers. Because of its general design, it should also be effective against any evolutionary enhancements to fingerprint scanners. It can protect an entire network against scans designed to profile vulnerable systems. Such scans are often the first step in an attack to gain control of exploitable computers. Once compromised these systems could be used as part of a distributed denial of service attack. By blocking the first step, the fingerprint scrubber increases the security of a heterogeneous network.

Acknowledgments

The Intel Corporation provided support for this work through a generous equipment donation and gift. This work was also supported in part by a research grant from the Defense Advanced Research Projects Agency, monitored by the U.S. Air Force Research Laboratory under Grant F30602-99-1-0527.

References

[1] F. Baker. Requirements for IP Version 4 Routers. RFC 1812, 1995.

[2] D. Brent Chapman and Elizabeth D. Zwicky. *Building Internet Firewalls*. O'Reily and Associates, Inc., 1995.

[3] Fyodor. Remote OS detection via TCP/IP stack fingerprinting. `http://www.insecure.org/nmap/nmap-fingerprinting-article.html`, October 1998.

[4] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, November 1998.

[5] Intel Internet Exchange Architecture. `http://developer.intel.com/design/IXA/`.

[6] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[7] G. Robert Malan, David Watson, Farnam Jahanian, and Paul Howell. Transport and Application Protocol Scrubbing. In *Proceedings of the IEEE INFOCOM 2000 Conference*, Tel Aviv, Israel, March 2000.

[8] David Maltz and Pravin Bhagwat. TCP Splicing for Application Layer Proxy Performance. Technical Report RC 21139, IBM Research Division, March 1998.

[9] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet Security Association and Key Mangement Protocol (ISAKMP). RFC 2408, November 1998.

[10] Ryan McCabe. Iplog. `http://ojnk.sourceforge.net/`.

[11] Netperf: A Network Performance Benchmark. `http://www.netperf.org/`.

[12] Vern Paxson. Automated packet trace analysis of TCP implementations. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[13] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Originally Secure Networks, Inc., now available as a white paper at the Network Associates Inc. homepage at `http://www.nai.com/`, January 1998.

[14] David Putzolu, Sanjay Bakshi, Satyendra Yadav, and Raj Yavatkar. The Phoenix Framework: A Practical Architecture for Programmable Networks. *IEEE Communications*, 38(3):160–165, March 2000.

[15] Queso Homepage. `http://www.apostols.org/projectz/queso/`.

[16] PGP Security. Gauntlet Firewall. `http://www.pgp.com/asp_set/products/tns/gauntlet.asp`.

[17] Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP Forwarder Performance. Technical Report TR98-01, Dept. of Computer Science, University of Arizona, February 1998.

[18] Trusted Information Systems. TIS Firewall Toolkit. `ftp://ftp.tislabs.com/pub/firewalls/toolkit`.

A Chosen Ciphertext Attack Against Several E-Mail Encryption Protocols

Jonathan Katz* Bruce Schneier†

Abstract

Several security protocols (PGP, PEM, MOSS, S/MIME, PKCS#7, CMS, etc.) have been developed to provide confidentiality and authentication of electronic mail. These protocols are widely used and trusted for private communication over the Internet. We point out a potentially serious security hole in these protocols: any encrypted e-mail can be decrypted using a one-message, adaptive chosen-ciphertext attack which exploits the structure of the block cipher chaining modes used. Although such attacks seem to be of primarily theoretical interest, we argue that they are feasible in the networked systems in which these e-mail protocols are used. We suggest several solutions to protect against this class of attack.

1 Introduction

Electronic mail (e-mail) has become an essential communication tool. The ease of e-mail communication, when compared to traditional choices such as physical mail, fax, or telephone, makes it the communications medium of choice for many people. As more people and businesses move on-line, and Internet access becomes more commonplace, e-mail is expected to become even more important as a communications tool.

In order for e-mail to fully supplant other alternatives, businesses (and, to a lesser extent, individual users) must be assured of the privacy of their e-mail correspondence. This is of special concern in the case of e-mail, since it is no doubt easier for an adversary to "tap" into an Internet link than to tap into a phone line. Furthermore, e-mail has the potential of offering security beyond that of telephone conversations, as there is currently no good way of "scrambling" a telephone conversation (although one can detect — and with a bit more trouble, prevent — the "tapping" of the phone line in the first place). It is primarily for these reasons that e-mail encryption protocols were developed.

In any system, there are multiple points which an adversary can attack; of course, a system is only as secure as its weakest point of attack. In this paper, we point out a so-called *chosen ciphertext* attack which succeeds against all current implementations of e-mail security protocols. Furthermore, we argue that this attack is entirely feasible in the networked environment in which these e-mail security protocols are used.

2 Background

2.1 E-mail Encryption Protocols

The attack outlined herein is applicable to many different e-mail encryption protocols [21, 22]. In this paper, we explicitly consider attacks on OpenPGP [7, 10, 24] (the attack is also applicable to previous versions of PGP), S/MIME [20] (building upon CMS [14] and/or PKCS#7 [15]), PEM [18], and MOSS [8]. We refer the reader to the listed references for an in-depth description of these protocols; in this paper, we merely provide a high-level description necessary

*Department of Computer Science, Columbia University; jkatz@cs.columbia.edu.

†Counterpane Internet Security, Inc.; schneier@counterpane.com.

for a proper understanding of the attack. Specifically, the attack exploits the symmetric-key modes of encryption used, and therefore only this detail of the encryption protocols (which is the same for all protocols, at this level of description) is presented.

Consider an e-mail message (or file) $M = M_1, M_2, \ldots$, where the message M is broken into a sequence of blocks of appropriate length for the underlying block cipher used (e.g. 64 bits for DES, 128 bits for AES). The protocols considered encrypt the message as follows:

1. A random "session-key" K is generated.

2. M is encrypted using a symmetric-key encryption algorithm (block cipher) and key K, using some mode of encryption (we discuss below the details of the mode used). This gives ciphertext $C_0, C_1, C_2, \ldots$ (note the generation of the additional ciphertext block C_0).

3. The session-key K is encrypted using the recipient's public key. This is represented by $\mathcal{E}_{\mathbf{pk}}(K)$.

4. The following message is sent to the recipient: $< \mathcal{E}_{\mathbf{pk}}(K), C_0, C_1, \ldots >$.

The recipient, reversing the above steps, uses his private key to compute K; given K, the recipient can then use symmetric-key decryption to determine the original message M.

Note that a mode of encryption (in step 2, above) is necessary in order to allow for the encryption of messages which are longer than one block.

2.2 Chosen Ciphertext Attack

The attack presented here is known in the cryptographic literature as an adaptive chosen-ciphertext attack. The reader is referred elsewhere for formal definitions [2, 16], but a simple description is provided here. Assume an adversary intercepts ciphertext C and is trying to determine the underlying plaintext $P = \mathcal{D}(C)$ (where $\mathcal{D}(\cdot)$ refers to decryption of the ciphertext). We refer to C as the *challenge ciphertext.* Under an adaptive[1] chosen-ciphertext attack, the adversary may submit ciphertexts $C_1, C_2, \ldots$ of his choice to a *decryption oracle* which then returns the corresponding plaintexts $P_1 = \mathcal{D}(C_1), P_2 = \mathcal{D}(C_2), \ldots$. The adversary is allowed to use the information thus obtained to recover the desired plaintext P. Note that for the attack to be non-trivial the adversary is not allowed to submit the challenge ciphertext C to the decryption oracle.

At first glance, this type of attack seems purely theoretical (when does an adversary have access to free decryption?), but consideration of the attack is of practical significance [12, 6, 4, 23, 13]. One can readily think of examples in which an adversary submitting a ciphertext to a user might obtain partial information about the decrypted plaintext. For instance, an adversary might be interacting with a computer which, when given some ciphertext, performs a specified action if and only if the ciphertext is *valid* (i.e., whose decryption corresponds to *some* plaintext); this allows an adversary to distinguish valid ciphertexts from invalid ones. Such an attack on the RSA Encryption Standard PKCS#1 has been demonstrated [4], leading to a feasible attack on certain implementations of the SSL V.3.0 protocol. A similar attack, called a "reaction attack," has been used to break several coding-theory based public-key cryptosystems [13]. In yet another example [12], the adversary communicates with a party on the network who responds to ciphertext messages only if the decryption of the message corresponds to valid English text. This, too, gives the adversary information about the decrypted plaintext which may prove useful in cryptanalysis.

3 Details of the Attack

We stress that we do not expose any weaknesses in the public-key (typically RSA or ElGamal) or symmetric-key (IDEA, CAST, or 3-DES) algorithms

[1] ***Adaptivity*** **in this context means that the adversary is allowed to submit ciphertexts to the decryption oracle even after viewing the challenge ciphertext. In a non-adaptive attack, the adversary may only submit ciphertexts *before* viewing the challenge ciphertext.**

used in the various encryption protocols. In fact, the attack is independent of the encryption algorithms used, and we therefore omit mention of specific algorithms when describing the attack. Rather, our attack focuses on the chaining mode used when encrypting messages longer than one block.

We begin with a description of the attack on the chaining mode used by OpenPGP [7]. OpenPGP uses a slight variation of Cipher Feedback (CFB) mode for symmetric encryption. Before encryption, the message M' is prepended by a 10-octet string. The first 8 octets are random, and the 9th and 10th octets are copies of the 7th and 8th octets, respectively. The resulting text $M = M_1, M_2, \ldots, M_k$ is parsed as a sequence of k blocks, each 64 bits long (for convenience, we assume a block cipher with 64-bit block size; the attack is similar for block ciphers with other block sizes). The OpenPGP variant of CFB-64 chaining mode is as follows ($\mathcal{E}_K(\cdot)$ represents application of the block cipher using session-key K):

Encryption: $c_0 = 0^{64}$
for $i = 1$ to k:
$c_i = M_i \oplus \mathcal{E}_K(c_{i-1})$
Output: $C_0, C_1, \ldots, C_k$

Decryption: for $i = 1$ to k:
$M_i = C_i \oplus \mathcal{E}_K(C_{i-1})$
if ($C_0 = 0^{64}$) and (9th and 10th octets match 7th and 8th octets)
Output: $M_1, M_2, \ldots, M_k$
else
Error

Given ciphertext $<\mathcal{E}_{\mathrm{pk}}(K), 0^{64}, C_1, \ldots, C_k>$, to obtain the value of block M_i ($i > 2$) one does the following:

1. Choose a (random) 64-bit number r.
2. Submit the ciphertext:
$< \mathcal{E}_{\mathrm{pk}}(K), 0^{64}, C_1, C_2, C_{i-1}, r >$.
3. Receive back the decryption $M' = M'_1, \ldots, M'_4$, where $M'_4 = r \oplus \mathcal{E}_K(C_{i-1})$.
4. Compute $M_i = M'_4 \oplus r \oplus C_i$.

(Note that this also allows determination of block M_2.) If there is concern that the decrypted message will appear too similar to the original message, a random string can be inserted between C_2 and C_{i-1}; also, note that the last 6 octets of C_2 can be randomly chosen.

Other chosen ciphertext attacks are also possible. For example, submitting:

$$< \mathcal{E}_{\mathrm{pk}}(K), 0^{64}, C_1, C_2^{16} r_1, C_3, r_2, \ldots, C_k, r_{k-1} >,$$

where C_2^{16} represents the first 16 bits of C_2, r_1 is a random 48-bit string, and $r_2, \ldots, r_{k-1}$ are random 64-bit strings, allows the adversary to compute the entire contents of the original message. The feasibility of any particular attack depends on the specifics of the "decryption oracle access" assumed available (which depends upon the behavior of the recipient of the original message; see Section 4).

This type of attack is not limited to protocols using CFB mode. CBC mode, used by PEM [18] and CMS [14], is also vulnerable to a chosen ciphertext attack, as are all other "popular" modes of encryption [22] (including ECB, OFB, PCBC, and counter mode). Note further that, due to the reliance of CBC and CFB modes on XOR operations, individual bits of selected plaintext blocks can be "set" at will by an adversary mounting an adaptive chosen ciphertext attack. In particular, this allows the adversary to circumvent any redundancies required by protocols using these modes.

4 Feasibility of the Attack

A chosen ciphertext attack in the context of e-mail encryption is certainly feasible. Imagine a situation in which *User* has configured his e-mail handler to automatically decrypt any incoming e-mails encrypted using PGP. An adversary *Adv* intercepts a PGP-encrypted message C sent to *User*, and wants to determine the contents of this message. Adversary *Adv* constructs C' according to the above algorithm, and sends this message to *User*. Then, *User*'s e-mail handler automatically decrypts C', and *User* reads the corresponding message M'. To *User*, the message appears garbled; he therefore replies to *Adv* with, for

example, "What were you trying to send me?", but also *quotes the "garbled" message M'*. *Adv* receives the plaintext M' which he wanted, and can use this to determine the original message. If *User* does not send *Adv* the garbled message, *Adv* can request it (for example: "I don't know what happened. Send me the file you decrypted and I'll try to figure it out."). This is not an unreasonable feat of social engineering. Note that this attack works even if all e-mail sent by *User* is encrypted[2]; when *User* responds to *Adv*'s garbled message, he encrypts his response using *Adv*'s public key (and thus *Adv* has access to this response, anyway).

In the setting described above it is important that M' not look too similar to M, because otherwise the original recipient may become suspicious. For example, the adversary will likely gain nothing if the submitted ciphertext decrypts to a plaintext which is 90% identical to the original message (which is technically allowed under a chosen ciphertext attack); in this case, the original recipient will certainly realize that something strange is going on.

Sometimes, messages are compressed before being encrypted [7]. This makes the attack more difficult for an adversary (due to the redundancy a valid compressed message must contain), but by no means precludes such an attack. Recall that the modes of operation used for encryption are "local" (in the sense that a plaintext block is affected by only very few nearby ciphertext blocks) and therefore the redundancy of the plaintext can be maintained via suitable alteration of the ciphertext. Furthermore, since the compression function is reversible and publicly known, the chosen ciphertext attack given above can be modified for this setting without difficulty.

5 Recommendations

We can immediately suggest some possible ways to prevent the attacks outlined herein. The simplest solution is for users not to reply to "garbage" e-mails by quoting the garbage message in their reply. This seems quite limiting — what if the message legitimately got corrupted in transit? It also relies too heavily on the behavior of users of the system.

Another solution is to demand that all encrypted messages be signed, and to not respond to unsigned messages with quotes from those messages. This is also limiting — it is common for people to send encrypted-but-unsigned e-mail — and the extra computational overhead should be avoided. It also shares many of the problems of the previous solution.

Yet another solution is to have the e-mail decryption software store all session keys which have been used so far in messages sent to the user. (Actually, the program should store a one-way hash of each session key, to avoid the problem of someone breaking into the user's files and obtaining a list of previously-used session keys.) Note that the chosen ciphertext attack described in Section 3 sends a previously-used session key as part of the ciphertext (in fact, it uses the same public-key encryption of the session key). The probability of this happening by chance is extremely low. A warning could be generated whenever a session key is repeated, to alert the user to be careful when responding to that particular message. Again, this solution is not completely satisfying. If encrypted e-mail becomes ubiquitous, storage of all session keys used becomes cumbersome.

A first step, and one which at least eliminates the "simple" attacks outlined in this paper, is for protocols to use a mode of encryption which is itself secure under adaptive chosen ciphertext attack [19, 5, 17]. Unfortunately, the various modes currently known to be secure in this setting each have their own drawbacks: expansion of the ciphertext length [17], or requirement of additional primitives [19] and multiple keys [19, 5] (which must then be encrypted using slow public-key methods). We hope that this paper will encourage additional research toward *efficient* one-key modes of encryption secure under chosen ciphertext attacks[3].

However, even this would not be sufficient to guarantee full security against an arbitrary adaptive chosen ciphertext attack. An adversary could mount a

[2] After all, the assumption is that this user is security conscious in the first place!

[3] A recently suggested mode, integrity-aware CBC (iaCBC), was subsequently broken. No provably-secure fix is currently known [11].

chosen ciphertext attack against the public-key algorithm itself, or exploit some interaction between the public-key algorithm and the block cipher being used (although no such attacks are known at present). Recent work on chosen-ciphertext secure hybrid encryption schemes [1, 9] (where *hybrid* implies integration of public-key and symmetric-key [i.e., block cipher] encryption) can be used achieve this goal. Unfortunately, currently proposed solutions are only heuristically secure, in that they are proven secure in a model in which a hash function is treated as a random function. More work in this direction is needed.

Another possibility (if one is willing to consider multiple-key approaches) is to generate two session keys: one for encryption and one for authentication. One then encrypts both keys using a public-key algorithm, encrypts the message using the first key (as before), and additionally appends a message authentication code (using the second key) of the ciphertext. Various approaches along these lines have been considered, and an exact security analysis of these approaches appears elsewhere [3].

References

[1] M. Abdalla, M. Bellare, and P. Rogaway, "DHAES: An Encryption Scheme Based on the Diffie-Hellman Problem," Manuscript, September 1998. Available at `http://www.cs.ucdavis.edu/~rogaway`.

[2] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations Among Notions of Security for Public-Key Encryption Schemes," *Advances in Cryptology — CRYPTO '98 Proceedings*, Springer-Verlag, 1998, pp. 26–45.

[3] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations Among Notions and Analysis of the Generic Composition Paradigm," Manuscript, May 2000. Available at `http://eprint.iacr.org`.

[4] D. Bleichenbacher, "Chosen Ciphertext Attacks Against Protocols Based on RSA Encryption Standard PKCS#1," *Advances in Cryptology — CRYPTO '98 Proceedings*, Springer-Verlag, 1998, pp. 1–12.

[5] D. Bleichenbacher and A. Desai, "A Construction of a Super-Pseudorandom Cipher," Manuscript, February 1999.

[6] M. Blum, P. Feldman, and S. Micali, "Proving Security Against Chosen Ciphertext Attacks," *Advances in Cryptology — CRYPTO '88 Proceedings*, Springer-Verlag, 1990, pp. 256–268.

[7] J. Callas, L. Donnerhacke, M. Finney, and R. Thayer, "OpenPGP Message Format," RFC 2440, Nov 1998.

[8] S. Crocker, N. Freed, J. Galvin, and S. Murphy, "MIME Object Security Services," RFC 1848, Oct 1995.

[9] E. Fujisaki and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes," *Advances in Cryptology — CRYPTO '99 Proceedings*, Springer-Verlag, 1999, pp. 537–554.

[10] S. Garfinkel, *PGP: Pretty Good Privacy*, O'Reilly & Associates, 1995.

[11] V. Gligor, Personal Communication, March 2000.

[12] S. Goldwasser, S. Micali, and P. Tong, "Why and How to Establish a Private Code on a Public Network," *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science* 1982, pp. 134–144.

[13] C. Hall, I. Goldberg, and B. Schneier, "Reaction Attacks Against Several Public-Key Cryptosystems," *Proceedings of Information and Communication Security*, Springer-Verlag, 1999, pp. 2–12.

[14] R. Housley, "Cryptographic Message Syntax," RFC 2630, Jun 1999.

[15] B. Kaliski, "PKCS #7: Cryptographic Message Syntax, Version 1.5," RFC 2315, Mar 1998.

[16] J. Katz and M. Yung, "Complete Characterization of Security Notions for Probabilistic Private-Key Encryption," *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing* 2000.

[17] J. Katz and M. Yung, "Unforgeable Encryption and Chosen-Ciphertext Secure Modes of Operation," *Fast Software Encryption — FSE '00 Proceedings*, Springer-Verlag, 2000.

[18] J. Linn, "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures," RFC 1421, Feb 1993.

[19] M. Naor and O. Reingold, "On the Construction of Pseudo-Random Permutations: Luby-Rackoff Revisited," *Journal of Cryptology* 12(1): 29-66 (1999).

[20] B. Ramsdell, "S/MIME Version 3 Message Specification," RFC 2633, June 1999.

[21] B. Schneier, *E-Mail Security*, John Wiley & Sons, 1995.

[22] B. Schneier, *Applied Cryptography, 2nd Edition*, John Wiley & Sons, 1996.

[23] V. Shoup, "Why Chosen Ciphertext Security Matters," IBM Research Report RZ 3076, November, 1998.

[24] P. Zimmerman, *The Official PGP User's Guide*, MIT Press, 1995.

PGP in Constrained Wireless Devices

Michael Brown* Donny Cheung* Darrel Hankerson† Julio Lopez Hernandez‡
Michael Kirkup* Alfred Menezes*

Abstract

The market for Personal Digital Assistants (PDAs) is growing at a rapid pace. An increasing number of products, such as the PalmPilot, are adding wireless communications capabilities. PDA users are now able to send and receive email just as they would from their networked desktop machines. Because of the inherent insecurity of wireless environments, a system is needed for secure email communications. The requirements for the security system will likely be influenced by the constraints of the PDA, including limited memory, limited processing power, limited bandwidth, and a limited user interface.

This paper describes our experience with porting PGP to the Research in Motion (RIM) two-way pager, and incorporating elliptic curve cryptography into PGP's suite of public-key ciphers. Our main conclusion is that PGP is a viable solution for providing secure and interoperable email communications between constrained wireless devices and desktop machines.

1 Introduction

It is expected that there will be more than 530 million wireless subscribers by the year 2001, and over a billion by 2004 (see [46]). Efforts are underway, most notable among them the Wireless Application Protocol (WAP) [50], to define and standardize the emerging wireless Internet. Users will access wireless services including telephony, email and web browsing, using a variety of wireless devices such as mobile phones, PDAs (such as the PalmPilot), pagers, and laptop computers equipped with wireless modems. Many wireless devices are constrained by limited CPU, memory, battery life, and user interface (e.g., small screen size, or a lack of graphics capabilities). Wireless networks are constrained by low bandwidth, high latency, and unpredictable availability and stability. The purpose of this paper is to examine the viability of using PGP for providing secure and interoperable email communications between constrained wireless devices and desktop machines.

There are two popular standards for email security: S/MIME and PGP. S/MIME [40] provides confidentiality and authentication services to the MIME (Multipurpose Internet Mail Extensions) Internet email format standard. PGP (Pretty Good Privacy) [8, 16] is an email security standard that has been widely used since it was first introduced by Zimmermann in 1991 [52]. While it appears that S/MIME will emerge as the industry standard for commercial and organizational use, it also appears that PGP will remain the choice for personal email security for many users in the years to come.

The specific goals of this project were three-fold:

1. Port the basic PGP functionality to the RIM pager, and implement a workable key management system and a usable user interface that is appropriate for the RIM pager environment.
2. Achieve interoperability with existing PGP implementations for workstation and PalmPilot platforms.
3. Incorporate standards-based and commercial-strength elliptic curve cryptography into PGP's suite of public-key algorithms.

The remainder of this paper is organized as follows. §2 provides a brief history of PGP, and summarizes the security services offered by PGP. A description of the RIM two-way pager including hardware, software, user interface, development tools, and the paging environment, is provided in §3. A brief overview of the PalmPilot is presented in §4. Elliptic curve cryptography is introduced in §5, along with a description of our implementation. We provide timing comparisons of our ECC implementation with RSA and DL implementations on a variety of platforms. Our experience with porting PGP to the RIM pager is described in §6. Our implementation, including a description of the user interface and key management facilities, is presented in §7. In §8, we describe some possible directions for future work. Finally, §9 makes concluding remarks.

*Dept. of Combinatorics and Optimization, University of Waterloo, Canada. Emails: {mk3brown, dccheung, mkirkup, ajmeneze}@uwaterloo.ca

†Dept. of Discrete and Statistical Sciences, Auburn University, USA. Email: hankedr@mail.auburn.edu. Supported by a grant from Auburn University COSAM.

‡Institute of Computing, State University of Campinas, Brazil, and Dept. of Computer Science, University of Valle, Colombia. Email: julioher@dcc.unicamp.br

2 Pretty Good Privacy

2.1 History of PGP

The history of the Pretty Good Privacy (PGP) application is both interesting and convoluted, and encompasses issues in national security, personal privacy, patents, personalities, and politics; see, for example, [16]. A myriad of PGP releases emerged, in part due to US Government restrictions on exports.

The initial PGP application was released in 1991. According to [16] this was an "emergency release" prompted in part by a proposed anti-crime bill which would require eavesdropping ability for the US Government on all communications systems. An RSA-based public-key scheme was used, along with a symmetric-key algorithm developed by Zimmermann known as Bass-O-Matic.

Security concerns over Bass-O-Matic resulted in its replacement with IDEA in PGP 2. A commercial version of PGP was developed in 1993 with ViaCrypt (which had a license from Public Key Partners for RSA). Although RSA Data Security had released a reference implementation (RSAREF) of RSA that could be used for non-commercial purposes, there were interface and other difficulties preventing its use in PGP. In 1994, RSAREF 2.0 was released and included changes which MIT recognized would solve the interface problems. This eventually led to PGP 2.6, a version which could be used freely for non-commercial purposes, and which quickly leaked out of the US and developed into several international variants.

MIT PGP 2.6.2 increased the ceiling on the maximum size of an RSA modulus (from 1024 to 2048 bits, although ViaCrypt reports a patch correcting certain bugs with the longer moduli). The symmetric-key cipher is IDEA, a 64-bit block cipher with 128-bit keys; MD5 is used as the hash function, having digest length of 128 bits. A dependency tree for various US and international versions and variants may be found via [38].

Work on PGP 3 began in 1994, and was released by PGP Inc (formed by Zimmermann) as PGP 5 in May 1997.[1] New algorithms were present, including DSA [34] for signatures, an ElGamal public-key encryption scheme [12], the Secure Hash Algorithm (SHA-1) [35] with 160-bit message digests, and the symmetric-key ciphers CAST and Triple-DES (64-bit block ciphers with key sizes of 128 and 168 bits, respectively).

In August of 1997, the IETF was approached concerning a proposal to bring PGP to a standards body as a protocol. An OpenPGP working group was formed. Using PGP 5 as the base, a format specification was promoted to a Proposed Standard by the IESG in October 1998. The resulting IETF specification for OpenPGP [9] describes an unencumbered architecture, although compatibility with PGP 2.6 was encouraged. A reference implementation was written by Tom Zerucha and provided in a form suitable for scanning to circumvent US export restrictions [8].

In December 1999, Network Associates (which had acquired PGP Inc in December 1997) was granted a license by the US Government to export PGP. An international PGP project [25], which had been making PGP available world-wide by scanning paper copies that were (legally) exported from the US, announced that the lifting of the ban on strong encryption "marks the end of the PGPi scanning and OCR project, which started with PGP 5.0i in 1997."

Several OpenPGP-compliant applications have been developed. The reference implementation by Zerucha [8] relies on the OpenSSL library [37], and has been used by Zerucha as the basis for a PalmPilot implementation. The standard does not require the use of patented algorithms, and applications such as GNU Privacy Guard [18], released in 1999 as a replacement for PGP, can be both compliant and distributable without patent restrictions (since it does not include IDEA or RSA).

2.2 PGP security services

KEY GENERATION AND STORAGE. PGP allows a user to generate multiple key pairs (public-key/private-key pairs) for each public scheme supported. Different key pairs are generated for public-key encryption and for digital signatures. The key pairs, together with public keys of other users, are stored in a file called the key ring.

Information stored with a public key includes the user's name, email address, trust and validity indicators, key type, key size, expiry date, fingerprint (e.g., the 160-bit SHA-1 hash of the formatted public key), and a key ID (e.g., the low order 64 bits of the fingerprint).

Private keys are not stored directly in the key ring. Instead, the user selects a passphrase which is salted and hashed to derive a key k for a symmetric encryption scheme. The private key is encrypted using k, the passphrase is discarded, and the encrypted private key is stored. Subsequently, when the user wishes to access a private key (in order to decrypt a message or sign a message), the passphrase must be supplied so that the system can regenerate k and recover the private key.

CRYPTOGRAPHIC SERVICES. PGP uses a combination of symmetric-key and public-key methods to provide authentication and confidentiality.

A message can be signed using the private key from a suitable public-key signature scheme. The recipient can

[1] Callas [8] notes that ViaCrypt had released several products with a version number of 4 although they were derivatives of PGP 2, and "it was easier to explain why three became five than to explain why three was the new program and four the old one."

verify the signature once an authentic copy of the signer's corresponding public key is obtained. The OpenPGP standard requires support for SHA-1 as a hash algorithm and the DSA, and encourages support for the MD5 hash function and RSA as a signature algorithm.

The use of symmetric-key algorithms (such as DES) alone for encryption is supported, although PGP is known more for the confidentiality provided by a combination of public-key and symmetric-key schemes. Since public-key encryption schemes tend to be computationally expensive, a session key is used with a symmetric-key scheme to encrypt a message; the session key is then encrypted using one or more public keys (typically, one for each recipient), and then the encrypted message along with each encrypted session key is delivered. The standard requires support for an ElGamal public-key encryption scheme and Triple-DES; support for RSA, IDEA, and CAST is encouraged.

Signatures and encryption are often used together, to provide authentication and confidentiality. The message is first signed and then encrypted as described above.

KEY MANAGEMENT. The OpenPGP standard does not have a trust model. An OpenPGP-compliant PGP implementation could support a hierarchical X.509-based public key infrastructure (PKI). The trust model employed by existing PGP implementations is a combination of direct trust and the web of trust. In the former, user A obtains B's public key directly from B; fingerprints facilitate this process as only the fingerprints have to be authenticated. In the web of trust model, one or more users can attest to the validity of B's public key by signing it with their own signing key. If A possesses an authentic copy of the public key of one of these users, then A can verify that user's signature thereby obtaining a measure of assurance of the authenticity of B's public key. This chaining of trust can be carried out to any depth.

3 RIM's Pager

3.1 Overview

The RIM wireless handheld device is built around a custom Intel 386 processor running at 10 MHz. Current models carry 2 Mbytes of flash memory and 304 Kbytes of SRAM. There is a fairly conventional (if rather small) keyboard with a 6- or 8-line by 28 character (depending on font) graphical display. A thumb-operated trackwheel takes the place of a conventional mouse (see Figure 1).

A set of applications including a calendar and address book are commonly installed; even the occasional game of Tetris (falling blocks) is possible with efficient use of the graphical display. The main attraction is the wireless communication features, in particular, email solutions. The integrated wireless modem is essentially invisible, with no protruding antennae. The device is roughly 3.5in x 2.5in x 1in (89mm x 64mm x 25mm) and weighs 5 ounces (142 g) with the single AA battery (there is also an internal lithium cell). RIM claims that the battery will last roughly three weeks with typical usage patterns.

A docking cradle can be used to directly connect the device to a serial port. Software for Microsoft Windows is provided to download programs and other information, and to synchronize application data. An RS-232 compatible serial port on the pager runs at 19200 bps.

To be slightly more precise, RIM has two hardware devices, the 850 and the 950, which are combined with software to provide communications solutions. We used RIM's BlackBerry solution [6] which uses the same hardware as the RIM Inter@ctive Pager 950. The 950 is more of a 2-way pager, sold in Canada by Cantel and in the US by BellSouth Wireless Data. The BlackBerry is sold directly by RIM and includes features such as single mailbox integration and PIM synchronization to the device.

The RIM 850 looks very similar to the 950 device, but runs on a different wireless network (ARDIS for the 850 as opposed to Mobitex for the 950). The RIM 850 is resold through American Mobile Satellite Corporation (AMSC) in the US, and is part of the AMSC and SkyTel eLink solution.

3.2 Software development

The BlackBerry Software Developer's Kit (SDK) is designed to make use of the features in Microsoft's C++ compiler packages. The SDK is freely available from [41]. A handheld application is built as a Windows DLL, a process which allows use of development and debugging facilities available for Windows. However, only a small subset of the usual library calls may be used, along with calls to SDK-supplied routines. The resulting DLL is then stripped of extraneous information and ported into the handheld operating system.

For simplicity, the multitasking is cooperative. An application is expected to periodically yield control; in fact, failure to yield within 10 seconds can trigger a pager reset. As an example, public-key operations tend to be computationally expensive, and it was necessary to insert explicit task yields in the code developed for this paper.

The SDK includes a simulator which can be used to test applications on the handheld operating system without having to download to the device (the images in this paper are snapshots of the simulator). A radio device (RAP modem) can be connected via serial port to the host machine so that applications running in the simulator can communicate with the Mobitex network. Alternately, a pager in the cradle can be used to exchange email with the simulator, provided that the pager is in coverage.

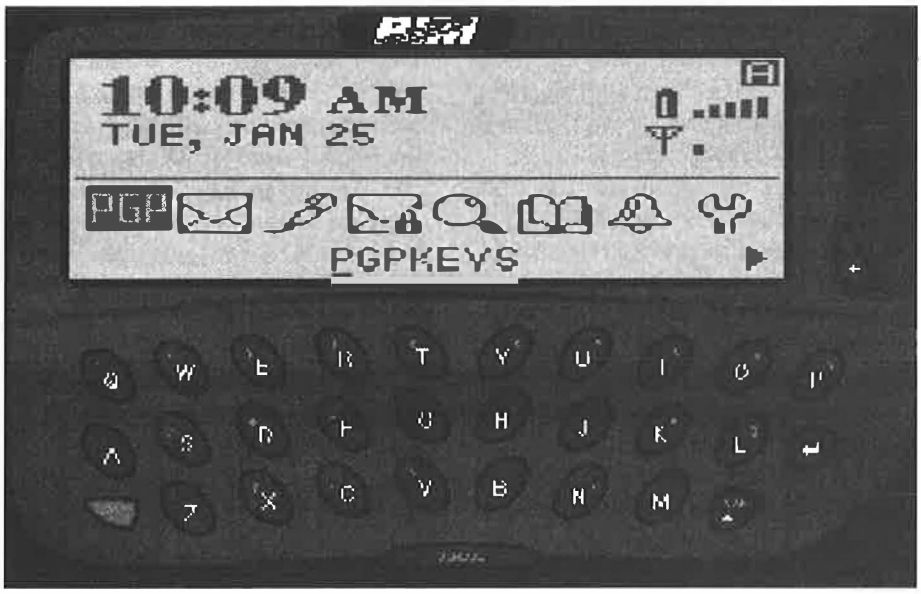


Figure 1: The RIM pager.

The simulator is essential for serious development, although testing on the pager can reveal bugs not found in the simulator. For example, we managed to link applications in such a way that they would work in the simulator but fail on the pager. At one point, we carelessly used some instructions introduced on the Intel 486, which would work in the simulator when running on a 486-or-better, but would fail on a 386.

3.3 File system

The pager relies on flash memory to store non-volatile data. Writing to flash is significantly more expensive than reading, primarily because flash is a write-once, bulk-erase device. Rewriting a single word of flash involves saving the contents of the 64K sector, erasing, and rewriting the entire sector. The longest step in this operation is erasing the sector, and takes approximately 5 seconds. A log-structured file system is employed in order to maintain acceptable performance. Periodically, the expensive process of committing the log updates is performed in order to free file system space.

The programming interface to the file system is generally through a relatively small number of high-level database-style calls. Handles are used to read and update databases and variable-length records, a simple but effective method to cooperate with the updating process of the log-structured file system. It is possible to use stream-style I/O operations of the type familiar to C programmers, which we occasionally found useful for testing code fragments developed on more traditional systems.

4 The PalmPilot

For comparison, our crypto routines were also run on the PalmPilot, a very popular PDA based on a 16 MHz Motorola 68000-type "Dragonball" processor.[2] Recent models carry 2–4 MB of memory in addition to ROM, although considerable expansion is possible. In 1999, wireless capabilities were introduced on the Palm VII. The communications model differs from the RIM device; in particular, the Palm does not qualify as a pager in the usual sense. There is an antenna which must be physically activated and then the device can request information. A NiCad battery charged from two AAA batteries common in the Palm series is used to power the radio.

Ian Goldberg had adapted portions of Eric Young's well-known SSLeay library (now OpenSSL [37]) for use on the PalmPilot [19]. The resulting library was used by Zerucha in building a Palm version of his reference OpenPGP, and by Daswani and Boneh [11] in their paper on electronic commerce.

We used Palm development tools based on the GNU C compiler (gcc-2.7.2.2). Timings were done on a Palm V running PalmOS 3.0. There are code segment and stack restrictions which must be considered in the design of a larger application, and our code had to be divided into several libraries in order to accommodate the Palm.

[2] According to [39], "Even after two rounds of Microsoft's best Windows CE efforts, PalmPilot OS devices still represent 80% of all palmtop sales."

5 Elliptic Curve Cryptography

5.1 Introduction

Elliptic curve cryptography (ECC) was proposed independently in 1985 by Neal Koblitz [27] and Victor Miller [33]. For an introduction to ECC, the reader is referred to Chapter 6 of Koblitz's book [29], or the recent book by Blake, Seroussi and Smart [7].

The primary reason for the attractiveness of ECC over RSA and discrete log (DL[3]) public-key systems is that the best algorithm known for solving the underlying hard mathematical problem in ECC (the elliptic curve discrete logarithm problem, ECDLP) takes fully exponential time. On the other hand, the best algorithms known for solving the underlying hard mathematical problems in RSA and DL systems (the integer factorization problem, and the discrete logarithm problem) take subexponential time. This means that the algorithms for solving the ECDLP become infeasible much more rapidly as the problem size increases than those algorithms for the integer factorization and discrete logarithm problems. For this reason, ECC offers security equivalent to that of RSA and DL systems, while using significantly smaller key sizes.

Table 1 lists ECC key lengths and very rough estimates of DL and RSA key lengths that provide the same security (against known attacks) as some common symmetric encryption schemes. The ECC key lengths are twice the key lengths of their symmetric cipher counterparts since the best general algorithm known for the ECDLP takes $(\sqrt{\pi 2^k})/2$ steps for k-bit ECC keys, while exhaustive key search on a symmetric cipher with l-bit keys takes 2^l steps. The estimates for DL security were obtained from [2]. The estimates for RSA security are the same as those for DL security because the best algorithms known for the integer factorization and discrete logarithm problems have the same expected running times. These estimates are roughly the same as the estimates provided by Lenstra and Verheul in their very thorough paper [31].

The advantages that may be gained from smaller ECC parameters include speed (faster computation) and smaller keys and certificates. These advantages are especially important in environments where processing power, storage space, bandwidth, or power consumption are at a premium such as smart cards, pagers, cellular phones, and PDAs.

[3] Examples of DL systems are the ElGamal public-key encryption scheme and the DSA signature scheme which is specified in the Digital Signature Standard. PGP documentation refer to these two schemes as Diffie-Hellman/DSS or DH/DSS.

5.2 Selecting ECC parameters

NOTATION. In the following, $\mathbb{F}_q$ denotes a finite field of order q, and E denotes an elliptic curve defined over $\mathbb{F}_q$. $\#E(\mathbb{F}_q)$ denotes the number of points on the elliptic curve E. The point at infinity is denoted by $\mathcal{O}$. There is a group law for adding any two elliptic curve points. If k is an integer and $P \in E(\mathbb{F}_q)$ is a point, then kP is the point obtained by adding together k copies of P; this process is called scalar multiplication.

DOMAIN PARAMETERS. ECC domain parameters consist of the following:

- q — the field size.
- FR — method used for representing field elements.
- a, b — elements of $\mathbb{F}_q$ which determine the equation of an elliptic curve E.
- G — the base point of prime order.
- n — the order of G.
- h — the cofactor: $h = \#E(\mathbb{F}_q)/n$.

The primary security parameter (see §5.4) is n. The ECC key length is thus defined to be the bitlength of n. Typical choices for q are an odd prime (in which case $\mathbb{F}_q$ is called a *prime field*) or a power of 2 (in which case $\mathbb{F}_q$ is called a *binary field*).

CURVES SELECTED. For this project, we chose binary fields $\mathbb{F}_{2^m}$, for $m = 163$, 233 and 283. Suitably chosen elliptic curves over these fields provide at least as much security as symmetric-key ciphers with key lengths 80, 112 and 128 bits respectively (see Table 1). A polynomial basis representation was used to represent field elements. Such a representation is defined by a reduction polynomial $f(x)$, which is an irreducible binary polynomial of degree m. For each field $\mathbb{F}_{2^m}$, we chose a random curve over $\mathbb{F}_{2^m}$ and a Koblitz curve [28] over $\mathbb{F}_{2^m}$ from the list of elliptic curves recommended by NIST for US federal government use [34]. The salient features of the Koblitz curves are provided in Table 2. Koblitz curves have special structure that enable faster elliptic curve arithmetic in some environments (see [44, 45]). The number of points on each of the chosen curves is almost prime; that is, $\#E(\mathbb{F}_{2^m}) = nh$, where n is prime and $h = 2$ or $h = 4$. Since $\#E(\mathbb{F}_{2^m}) \approx 2^m$, it follows that the ECC key length is approximately equal to m. Security implications of these choices are discussed in §5.4.

5.3 ECC protocols

KEY GENERATION. An entity A's public and private key pair is associated with a particular set of EC domain parameters $(q, \text{FR}, a, b, G, n, h)$. This association can be assured cryptographically (e.g., with certificates) or by context (e.g., all entities use the same domain parameters).

Symmetric cipher key lengths	Example algorithm	ECC key lengths for equivalent security	DL/RSA key lengths for equivalent security
80	SKIPJACK	160	1024
168	Triple-DES	224	2048
128	128-bit AES	256	3072
192	192-bit AES	384	7680
256	256-bit AES	512	15360

Table 1: ECC, DL, and RSA key length comparisons.

m	163
$f(x)$	$x^{163} + x^7 + x^6 + x^3 + 1$
E	$Y^2 + XY = X^3 + X^2 + 1$
n	4000000000000000000020108A2E0CC0D99F8A5EF
h	2
m	233
$f(x)$	$x^{233} + x^{74} + 1$
E	$Y^2 + XY = X^3 + 1$
n	8000000000000000000000000000069D5BB915BCD46EFB1AD5F173ABDF
h	4
m	283
$f(x)$	$x^{283} + x^{12} + x^7 + x^5 + 1$
E	$Y^2 + XY = X^3 + 1$
n	1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE9AE2ED07577265DFF7F94451E061E163C61
h	4

Table 2: Koblitz curves selected.

To generate a key pair, entity A does the following:

1. Select a random integer d from $[1, n-1]$.
2. Compute $Q = dG$.
3. A's public key is Q; A's private key is d.

PUBLIC KEY VALIDATION. This process ensures that a public key has the requisite arithmetic properties. A public key $Q = (x_Q, y_Q)$ associated with domain parameters $(q, \text{FR}, a, b, G, n, h)$ is validated using the following procedure:

1. Check that $Q \neq \mathcal{O}$.
2. Check that x_Q and y_Q are properly represented elements of $\mathbb{F}_q$.
3. Check that Q lies on the elliptic curve defined by a and b.
4. Check that $nQ = \mathcal{O}$.

The computationally expensive operation in public key validation is the scalar multiplication in step 4. This step can sometimes be incorporated into the protocol that uses Q – this is done in the ECAES below. Public key validation with step 4 omitted is called *partial* public key validation.

ELLIPTIC CURVE AUTHENTICATED ENCRYPTION SCHEME (ECAES). The ECAES, proposed by Abdalla, Bellare and Rogaway [1], is a variant of the ElGamal public-key encryption scheme [12]. It is efficient and provides security against adaptive chosen-ciphertext attacks.

We suppose that receiver B has domain parameters $D = (q, \text{FR}, a, b, G, n, h)$ and public key Q. We also suppose that A has authentic copies of D and Q. In the following, MAC is a message authentication code (MAC) algorithm such as HMAC [30], ENC is a symmetric encryption scheme such as Triple-DES. KDF denotes a key derivation function which derives cryptographic keys from a shared secret point.

To encrypt a message m for B, A does:

1. Select a random integer r from $[1, n-1]$.
2. Compute $R = rG$.
3. Compute $K = hrQ$. Check that $K \neq \mathcal{O}$.
4. Compute $k_1 \| k_2 = \text{KDF}(K)$.
5. Compute $c = \text{ENC}_{k_1}(m)$.
6. Compute $t = \text{MAC}_{k_2}(c)$.
7. Send (R, c, t) to B.

To decrypt ciphertext (R, c, t), B does:

1. Perform a partial key validation on R.
2. Compute $K = hdR$. Check that $K \neq \mathcal{O}$.
3. Compute $k_1 \| k_2 = \text{KDF}(K)$.
4. Verify that $t = \text{MAC}_{k_2}(c)$.
5. Compute $m = \text{ENC}_{k_1}^{-1}(c)$.

The computationally expensive operations in encryption and decryption are the scalar multiplications in steps 2-3 and step 2, respectively.

ELLIPTIC CURVE DIGITAL SIGNATURE ALGORITHM (ECDSA). The ECDSA is the elliptic curve analogue of the DSA [34]. SHA-1 is the 160-bit hash function [35].

We suppose that signer A has domain parameters $D = (q, \text{FR}, a, b, G, n, h)$ and public key Q. We also suppose that B has authentic copies of D and Q.

To sign a message m, A does the following:

1. Select a random integer k from $[1, n-1]$.
2. Compute $kG = (x_1, y_1)$ and $r = x_1 \bmod n$. If $r = 0$ then go to step 1.
3. Compute $k^{-1} \bmod n$.
4. Compute $e = \text{SHA-1}(m)$.
5. Compute $s = k^{-1}\{e + dr\} \bmod n$. If $s = 0$ then go to step 1.
6. A's signature for the message m is (r, s).

To verify A's signature (r, s) on m, B should do the following:

1. Verify that r and s are integers in $[1, n-1]$.
2. Compute $e = \text{SHA-1}(m)$.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5. Compute $u_1 G + u_2 Q = (x_1, y_1)$.
6. Compute $v = x_1 \bmod n$.
7. Accept the signature if and only if $v = r$.

The computationally expensive operations in signature generation and signature verification are the scalar multiplications in step 2 and step 5, respectively.

5.4 Security issues

HARDNESS OF THE ECDLP. It can easily be verified that the elliptic curves $E(\mathbb{F}_q)$ chosen resist all known attacks on the ECDLP. Specifically:

1. The number of points, $\#E(\mathbb{F}_q)$, is divisible by a prime n that is sufficiently large to resist the parallelized Pollard rho attack [36] against general curves, and its improvements [15, 48] which apply to Koblitz curves.
2. n does not divide $q^k - 1$ for all $1 \leq k \leq 30$, confirming resistance to the Weil pairing attack [32] and the Tate pairing attack [13].
3. $\#E(\mathbb{F}_q) \neq q$, confirming resistance to the Semaev attack [43].
4. All binary fields $\mathbb{F}_{2^m}$ chosen have the property that m is prime, thereby circumventing recent attacks [14, 17] on the ECDLP for elliptic curves over binary fields $\mathbb{F}_{2^m}$ where m is composite.

SECURITY OF ECAES. The ECAES modifies the ElGamal encryption scheme by using the one-time Diffie-Hellman shared secret, $hrdG$, to derive secret keys k_1 and k_2 The first key k_1 is used to encrypt the message using a symmetric cipher, while the second key k_2 is used to authenticate the resulting ciphertext. The latter provides resistance to chosen-ciphertext attacks. Some formal justification of ECAES security is provided in [1], where it is proven to be semantically secure against adaptive chosen-ciphertext attack on the assumption that the underlying symmetric encryption and MAC schemes are secure, and assuming the hardness of certain variants of the elliptic curve Diffie-Hellman problem.

In order to correctly balance the security of the ECAES cryptographic components, one should ideally employ a $\frac{k}{2}$-bit block cipher and a k-bit hash function for HMAC when using a k-bit elliptic curve (see Table 1). Our implementation used the 112-bit block cipher Triple-DES in CBC-mode and the 160-bit hash function SHA-1 for all 3 choices of ECC key lengths (163, 233 and 283). A future version of our implementation should allow for a variable output-length hash function (e.g., the forthcoming SHA-2) and a variable-length block cipher (e.g., the AES).

SECURITY OF ECDSA. ECDSA is the straightforward elliptic curve analogue of the DSA, which has been extensively scrutinized since it was proposed in 1991. For a summary of the security properties of the ECDSA, see [26].

Our implementation used the 160-bit hash function SHA-1 for all 3 choices of ECC key lengths (163, 233 and 283). As with the ECAES, a future version of our ECDSA implementation should allow for a variable output-length hash function.

5.5 Timings

This section presents timings for the ECC operations on a Pentium II 400 MHz machine, a PalmPilot and the RIM pager, and compares them with timings for RSA and DL operations.

ECC TIMINGS. Our ECC code was written entirely in C on a Sun Sparcstation and, in order to ensure porta-

bility, no assembler was used. We encountered no problems in porting the code to the Pentium II, RIM pager, and PalmPilot platforms, although some changes were required in order to cooperate with the 16-bit options used in the Palm version of the "big number" library of OpenSSL. No effort was made to optimize the ECC code for these particular platforms; it is very likely that significant performance improvements could be obtained by optimizing the ECC (and DL and RSA) code for these platforms. Further details of our ECC implementations are reported in [23].

For other ECC implementation reports, see [42] for a C implementation of elliptic curve arithmetic over $\mathbb{F}_{2^{155}}$, [49] for a C/C++ of elliptic curve arithmetic over $\mathbb{F}_{2^{191}}$ and over a 191-bit prime field, and [22] for an assembly language implementation of elliptic curve arithmetic over a 160-bit prime field on a 10 MHz 16-bit microcomputer.

Tables 3, 4 and 5 present timings of our implementation for ECC operations using the Koblitz curves and random curves over $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$.

RSA TIMINGS. The RSA code, written entirely in C, was taken from the OpenSSL library [37]. Tables 6 and 7 present timings for 512, 768, 1024, and 2048-bit RSA operations.

DL TIMINGS. The DSA and ElGamal code, also written entirely in C, was obtained from the OpenSSL and OpenPGP libraries. For ElGamal, the prime p was chosen to be a safe prime; that is $p = 2q + 1$ where q is also prime. Table 8 presents timings for 512, 768 and 1024-bit DSA and ElGamal operations. For encryption, the per-message secret key is not of full length (i.e., the bitlength of p), but of bitlength 200 + (bitlength of p)/32; this explains why ElGamal encryption is faster than ElGamal decryption. The ElGamal operations could be sped up significantly if DSA-like parameters were used (i.e., $p = kq + 1$, where q is a 160-bit prime).

COMPARISON. The performance of all three families of public-key systems (ECC, RSA and DL) are sufficiently fast for PGP implementations on a Pentium machine—it hardly matters whether a user has to wait 10 ms or 100 ms to sign and encrypt a message.

On the pager, RSA public-key operations (encryption and signature verification) are faster than ECC public-key operations, especially when the public exponent is $e = 3$. For example, verifying a 1024-bit RSA signature takes about 300 ms, while verifying a 163-bit ECC signature (using a Koblitz curve) takes about 1,800 ms. On the other hand, RSA private-key operations (decryption and signature generation) are slower than ECC private-key operations. For example, signing with a 1024-bit RSA key takes about 16,000 ms, while signing with a 163-bit ECC key takes about 1,000 ms. ECC has a clear advantage over RSA for PGP operations that require both private key and public key computations. Signing-and-encrypting together takes 16,400 ms with 1024-bit RSA (using $e = 3$), and 2800 ms with 163-bit ECC (using a Koblitz curve). Verifying-and-decrypting together takes 16,200 ms with 1024-bit RSA, and 2,900 ms with 163-bit ECC.

Similar conclusions are drawn when comparing RSA and ECC performance on the PalmPilot.

Private key operations with 2048-bit RSA are too slow for the pager and the PalmPilot, while 233-bit ECC and 283-bit ECC operations are tolerable for PGP applications on the pager.

Since domain parameters are used in our ECC implementation, ECC key generation only involves a single scalar multiplication and thus is very fast on the pager. RSA, ElGamal and DSA key generation on the pager is prohibitively slow. However, ElGamal and DSA key generation would be feasible on the pager if precomputed domain parameters (primes p and q, and generator g) were used.

5.6 Interoperability

The elliptic curves and protocols were selected to conform with the prevailing ECC standards and draft standards.

The Koblitz and random curves over $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$ are from the list of NIST recommended curves [34]. The representations, for both field elements and for elliptic curve points, are compliant with the ANSI X9.62 [4], ANSI X9.63 [5], IEEE P1363 [24] and FIPS 186-2 [34] standards. In addition, the Koblitz curve over $\mathbb{F}_{2^{163}}$ is explicitly listed in the WAP wTLS specification [51].

Our ECDSA implementation conforms to the security and interoperability requirements of ANSI X9.62, IEEE P1363, and FIPS 186-2. Our ECAES implementation conforms to the security and interoperability requirements of ANSI X9.63. The cryptographic components HMAC and Triple-DES (in CBC mode) of ECAES are compliant, respectively, with RFC 2104 [30] and ANSI X9.52 [3].

6 Porting PGP to the Pager

There are now a number of cryptographic libraries and PGP applications which have received extensive development and for which source code is available; see, for example, cryptlib by Peter Gutmann [20] and Crypto++ by Wei Dai [10]. Our plan was to adapt existing code, adding public-key schemes based on elliptic curves. For comparisons and development, it was essential that the

	Koblitz curve over $\mathbb{F}_{2^{163}}$			Random curve over $\mathbb{F}_{2^{163}}$		
	RIM pager	PalmPilot	Pentium II	RIM pager	PalmPilot	Pentium II
Key generation	751	1,334	1.47	1,085	1,891	2.12
ECAES encrypt	1,759	2,928	4.37	3,132	5,458	6.67
ECAES decrypt	1,065	1,610	2.85	2,114	3,564	4.69
ECDSA signing	1,011	1,793	2.11	1,335	2,230	2.64
ECDSA verifying	1,826	3,263	4.09	3,243	5,370	6.46

Table 3: Timings (in milliseconds) for ECC operations over $\mathbb{F}_{2^{163}}$ on various platforms.

	Koblitz curve over $\mathbb{F}_{2^{233}}$			Random curve over $\mathbb{F}_{2^{233}}$		
	RIM pager	PalmPilot	Pentium II	RIM pager	PalmPilot	Pentium II
Key generation	1,552	2,573	3.11	2,478	3,948	4.58
ECAES encrypt	3,475	5,563	7.83	6,914	11,373	13.99
ECAES decrypt	2,000	2,969	4.85	4,593	7,551	9.55
ECDSA signing	1,910	3,080	4.03	3,066	4,407	5.52
ECDSA verifying	3,701	5,878	7.87	7,321	11,964	14.08

Table 4: Timings (in milliseconds) for ECC operations over $\mathbb{F}_{2^{233}}$ on various platforms.

code run on several platforms in addition to the RIM device.

Our initial work was with GNU Privacy Guard (GnuPG) [18], an OpenPGP-compliant freely distributable replacement for PGP, which was nearing a post-beta release in 1999. Initial tests on the pager with several fragments adapted from GnuPG sources were promising, and the code appeared to be ideal for adding the elliptic curve routines and testing on Unix-based and other systems. However, it appeared that untangling code dependencies for our use on the pager would be unpleasant. (Perhaps a better understanding of GnuPG internals and design decisions would have changed our opinion.)

Jonathan Callas suggested that we look again at the OpenPGP reference implementation [8], which we had put aside after initial testing revealed a few portability and alignment problems in the code. The reference implementation relied on the OpenSSL library [37].

The OpenPGP reference implementation is surprisingly complete for the amount of code, although it is admittedly a little rough on the edges.[4] The code was developed on a Linux/x86 system, and modifications were required for alignment errors which prevented the program from running on systems such as Solaris/SPARC. In addition, some portability changes were required, including code involving the "long long" data type. For the RIM pager, the separation of the PGP code from the well-tested OpenSSL library, along with the small size of the OpenPGP sources, were definite advantages. Finally, it should be noted that the OpenSSL libraries build easily on Unix and Microsoft Windows systems, and are designed so that adding routines such as the elliptic curve code is straightforward.

Although applications for the pager are built as Windows DLLs, the pager is not a Windows-based system. There are significant restrictions on the calls that can be used, extending to those involving memory allocation, time and character handling, and the file system. There is no floating-point processor on the pager. In order to adapt code developed on more traditional systems, we wrote a library of compatibility functions to use with the pager. Some functions were trivial (such as those involving memory allocation, since the SDK included equivalent calls); others, such as the stream I/O calls, were written to speed testing and porting and cannot be recommended as particularly robust or elegant.

We used portions of OpenSSL 0.9.4, along with the library in the OpenPGP reference implementation. Relatively few changes to OpenSSL were required, and could be restricted to header files in many cases. The elliptic curve routines were integrated, including additions to the scripts used to build OpenSSL. For some platforms, OpenSSL can be built using assembly-language versions of certain key routines to improve execution speed. Some of these files for the Intel x86 include instructions (such as bswap) which were introduced for the 486, and cannot be used on the pager.

The OpenPGP sources were modified to correct the alignment bugs and portability problems mentioned above, and necessary changes were made for the elliptic curve schemes (public-key algorithms 18 and 19 in the

[4] Zerucha writes that he wasn't "careful about wiping memory and preventing memory leaks and other things to make the code robust" [8].

	Koblitz curve over $\mathbb{F}_{2^{283}}$			Random curve over $\mathbb{F}_{2^{283}}$		
	RIM pager	PalmPilot	Pentium II	RIM pager	PalmPilot	Pentium II
Key generation	2,369	4,062	4.50	3,857	6,245	6.88
ECAES encrypt	5,227	8,579	11.02	11,264	18,273	20.86
ECAES decrypt	2,932	4,495	6.78	7,498	12,046	13.88
ECDSA signing	2,760	4,716	5.64	4,264	6,816	8.08
ECDSA verifying	5,485	9,059	11.46	11,587	18,753	21.15

Table 5: Timings (in milliseconds) for ECC operations over $\mathbb{F}_{2^{283}}$ on various platforms.

	512-bit modulus			768-bit modulus		
	Pager	Pilot	Pentium II	Pager	Pilot	Pentium II
RSA key generation	73,673	189,461	346.77	287,830	496,356	953.01
RSA encrypt ($e = 3$)	213	317	1.13	388	587	1.87
RSA encrypt ($e = 17$)	262	410	1.28	451	753	2.17
RSA encrypt ($e = 2^{16} + 1$)	428	743	1.90	793	1,347	3.32
RSA decrypt	2,475	5,858	11.05	7,905	16,262	28.05
RSA signing	2,466	5,751	10.78	7,889	16,047	27.72
RSA verifying ($e = 3$)	99	200	0.40	214	413	0.78
RSA verifying ($e = 17$)	147	293	0.56	273	577	1.07
RSA verifying ($e = 2^{16} + 1$)	314	623	1.17	616	1,221	2.24

Table 6: Timings (in milliseconds) for 512-bit and 768-bit RSA operations on various platforms.

OpenPGP specification [9]). The compatibility library, along with a few stream-to-memory conversion functions allowed fairly direct use of the OpenPGP sources on the pager.

The only code tested exclusively in the pager environment involved the user interface (see §7.1). The SDK provides a fairly powerful and high-level API for working with the display and user input. The difficulties we encountered were mostly due to the lack of support in the API for direct manipulation of messages desired in a PGP framework. In part, this reflects a deliberate design decision by BlackBerry to develop a robust and intuitive communication solution which provides some protection against misbehaving applications.[5]

The pager DLLs for the interface and PGP library were over 400 KB in combined size. This includes all of the OpenPGP required algorithms and recommended algorithms such as IDEA and RSA, along with the new schemes based on elliptic curves. For a rough comparison, the code size for the main executable from the OpenPGP reference implementation (with the addition of the elliptic curve routines) is 300–400 KB, depending on platform.

[5] During our work on this project, BlackBerry modified the API to provide some of the access needed to smoothly integrate PGP into their mail application.

7 Implementation

7.1 User interface

PGP in any form has not been an easy application for novices to manage properly, in part due to the sophistication required, but also because of poor interface design [47]. The goals for our user interface design were rather modest: that a user who is familiar with using PGP on a workstation, and is comfortable operating the RIM device, should, without having to refer to a manual or help pages, be easily able to figure out how to use PGP on the pager and avoid dangerous errors (such as those described in [47]). As mentioned in §3.1, the graphics capabilities and screen size of the RIM device are very limited. This forced us to keep our PGP implementation simple and only offer the user the essential features.

A glimpse of our user interface is provided in Figures 1–5. Clicking on the PGP icon (see Figure 1) displays the list of users whose keys are in the public key ring (see Figure 2). Selecting a user name displays the menu shown in Figure 3, which allows the user to view the key's attributes, compose a new key, delete a key, or send a key.

	1024-bit modulus			2048-bit modulus		
	Pager	Pilot	Pentium II	Pager	Pilot	Pentium II
RSA key generation	580,405	1,705,442	2,740.87	—	—	26,442.04
RSA encrypt ($e = 3$)	533	1,023	2.70	1,586	3,431	7.26
RSA encrypt ($e = 17$)	683	1,349	3.23	2,075	4,551	9.09
RSA encrypt ($e = 2^{16} + 1$)	1,241	2,670	5.34	4,142	8,996	16.57
RSA decrypt	15,901	36,284	67.32	112,091	292,041	440.78
RSA signing	15,889	36,130	66.56	111,956	288,236	440.69
RSA verifying ($e = 3$)	301	729	1.23	1,087	2,392	4.20
RSA verifying ($e = 17$)	445	1,058	1.76	1,585	3,510	6.10
RSA verifying ($e = 2^{16} + 1$)	1,008	2,374	3.86	3,608	7,973	13.45

Table 7: Timings (in milliseconds) for 1024-bit and 2048-bit RSA operations on various platforms.

	512-bit modulus			768-bit modulus			1024-bit modulus		
	Pager	Pilot	PII	Pager	Pilot	PII	Pager	Pilot	PII
ElGamal key gen	—	—	51,704	—	—	219,820	—	—	1,200,157
ElGamal encrypt	7,341	17,338	19.13	16,078	34,904	35.91	26,588	73,978	67.78
ElGamal decrypt	8,704	19,060	22.55	26,958	56,708	59.53	57,248	148,059	144.73
DSA key gen	—	—	3,431	—	—	14,735	—	—	54,674
DSA signing	2,955	6,329	7.53	6,031	11,875	15.55	9,529	25,525	24.28
DSA verifying	5,531	12,389	14.31	11,594	24,277	26.13	18,566	52,286	47.23

Table 8: Timings (in milliseconds) for DL operations on various platforms.

Figure 2: Listing of PGP keys.

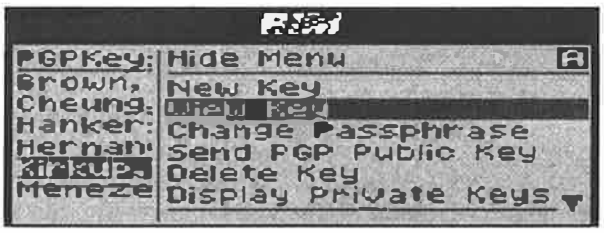


Figure 3: The main menu.

7.2 Key generation and storage

The main PGP menu (Figure 3) has an option "New Key" for creating a key pair. Users can enter their name, email address, pager PIN, and select a key type and key length (see Figure 4). The key types and key sizes presently available are ECC (random curve or Koblitz curve; over $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$ or $\mathbb{F}_{2^{283}}$), DH/DSS (512/512, 768/768, 1024/1024, 1536/1024 or 2048/1024 bits), and RSA (512, 768, 1024, 1536 or 2048 bits). The DH/DSS and RSA key sizes are the ones available in many existing PGP implementations. For the DSA, the maximum bitsize of the prime p is 1024 bits in conformance with the DSS [34]. For ECC, separate key pairs are generated for public-key encryption and digital signatures.

Public keys and private keys are stored in separate key rings. Public key attributes (see Figure 5) can be viewed using the "View Key" function available on the main menu. As required by OpenPGP, private keys are encrypted under a user-selected passphrase, and the encrypted private key is stored. The passphrase has to be entered whenever a private key is required to sign or decrypt a message.

7.3 Cryptographic services

The three basic PGP services are available: sign only, encrypt only, or sign-and-encrypt. Users can decide to sign an email, or to encrypt an email, after composing the message. The user is prompted for the passphrase to unlock the private signing key, and to select the public encryption key of the intended recipient. In addition to the times given in Tables 3–8 for the main operations, there is additional overhead which can be apparent to the

Figure 4: Screen for creating a new key pair.

Figure 5: Screen for viewing a (portion of the) public key's attributes.

user. Verifying the passphrase, for example, may require 20 seconds if the default iteration count is used when hashing the salted passphrase; our implementation used a smaller default iteration count. A small amount of time is added for interaction with the database filesystem for large memory transfers.

7.4 Key management

The key management system we implemented was the simplest one possible—the direct trust model (see §2.2). A menu item is available (see Figure 3) for emailing one's public key to another user. A function is also available for extracting and storing a public key received in an email message. If desired, a public key can be authenticated by verifying its fingerprint by some direct means (e.g., communicating it over the telephone—authenticity is provided by voice recognition).

8 Future Work

The following are some directions for future work.

RANDOM NUMBER GENERATION. Many systems implement a "random gathering device" which attempts to use environmental noise (keyboard data, system timers, disk characteristics, etc.) to build a cryptographically secure source of random bits [21]. Our pager application used only a rather simple (and most likely not sufficiently secure) seeding process involving the clock and a few other sources. A more sophisticated solution is essential, perhaps tapping into the radio apparatus as a source.

CODE SIZE. No serious effort was made to minimize the size of the programs loaded to the pager. There is some code linked from the OpenSSL cryptographic library which could easily be removed (in fact, we were somewhat surprised that the library with the added elliptic curve routines could be used with relatively few modifications for the pager). The library routines adapted from OpenSSL and OpenPGP along with various glue needed to adapt to the pager accounts for approximately 3/4 of the 370KB loaded on the device (with the remainder attributed to code involving the screen and user-interface). If some interoperability can be sacrificed, then the code size can also be reduced by removing routines such as CAST or some of the hash algorithms.

MAKING THE OPENPGP CODE MORE ROBUST. The OpenPGP reference implementation provides minimal diagnostics and can easily break on bad data. The occasional segmentation fault triggered by bad user data may be merely unpleasant when an application is used on a workstation; such errors on the pager are completely unacceptable. Our application corrects some of the most troublesome shortcomings, but better error-handling is needed.

KEY MANAGEMENT. We would like to implement an X.509-based PKI or the web of trust model. In either case, we would implement a key server for retrieving and storing keys in a key repository. This would involve setting up a proxy wireless server with which the pager would communicate directly. The proxy server in turn would communicate with existing key servers on the Internet.

9 Conclusions

IMPLEMENTING PGP ON THE RIM PAGER. The 32-bit architecture, relatively sophisticated operating system and development environment, and relatively large memory size means that development for the pager is closer to that done for more traditional systems than the small size might suggest. The user interface must be customized for the device, but "generic code" which does not involve file I/O moves fairly easily to the pager.

On the other hand, it appears likely that such devices will continue to have processors which run much more slowly than their desktop counterparts. Long delays in handling encrypted messages or signatures will be a considerable annoyance for users of this type of device. While we used a significant amount of the available memory on the pager, it would be desirable to reduce the resource consumption in a production version of PGP. Battery life will continue to be a major concern, and the overhead of authentication and confidentiality competes with the need to minimize transmissions from the device.

INTEROPERABILITY. The goal of interoperability was met. All of the required algorithms from RFC 2440 are included, along with several listed as recommended and the elliptic curve routines. Our PGP implementation interoperated with existing implementations for the PalmPilot and workstations.

ELLIPTIC CURVE CRYPTOGRAPHY. Elliptic curve solutions fit particularly well into the constrained environment. 1024-bit and 2048-bit RSA private-key operations are too slow for PGP applications, while the performance of 163-bit, 233-bit and 283-bit ECC operations is tolerable for PGP applications. If PGP (or other email security solutions) is to be used for securing email communications between constrained wireless devices and desktop machines, then our timings show that ECC is preferable to RSA since the performance of the latter on some wireless devices is too slow, while both systems perform sufficiently well on workstations.

GENERAL. This paper concentrated on PGP, although the results are more widely applicable. Many of the services targeted at the growing wireless market will require security solutions involving the cryptographic mechanisms used by PGP. The constraints on small wireless devices are likely to be with us for some time, and will require a balance of usability, computational requirements, security, and battery life.

Acknowledgements

The authors would like to thank Jonathan Callas for some enlightening discussions about PGP, and Herb Little for answering our numerous questions about the RIM pager.

References

[1] M. Abdalla, M. Bellare and P. Rogaway, "DHAES: An encryption scheme based on the Diffie-Hellman problem", preprint, 1999. Available from http://www.cs.ucdavis.edu/~rogaway/papers

[2] ANSI X9.30-1, "The digital signature algorithm (DSA) (revised)", American Bankers Association, working draft, July 1999.

[3] ANSI X9.52, "Triple data encryption algorithm modes of operation", American Bankers Association, 1998.

[4] ANSI X9.62, "The elliptic curve digital signature algorithm (ECDSA)", American Bankers Association, 1999.

[5] ANSI X9.63, "Elliptic curve key agreement and key transport protocols", American Bankers Association, working draft, August 1999.

[6] Blackberry, http://www.blackberry.net

[7] I. Blake, G. Seroussi and N. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.

[8] J. Callas, *OpenPGP Specification and Sample Code*, Printers Inc. Bookstore, Palo Alto, March 1999.

[9] J. Callas, L. Donnerhacke, H. Finney and R. Thayer, "OpenPGP message format", Internet RFC 2440, November 1998.

[10] W. Dai, Crypto++. http://www.eskimo.com/~weidai/cryptlib.html

[11] N. Daswani and D. Boneh, "Experimenting with electronic commerce on the PalmPilot", *Financial Cryptography '99*, Lecture Notes in Computer Science, **1648** (1999), Springer-Verlag, 1-16.

[12] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Transactions on Information Theory*, **31** (1985), 469-472.

[13] G. Frey and H. Rück, "A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves", *Mathematics of Computation*, **62** (1994), 865-874.

[14] S. Galbraith and N. Smart, "A cryptographic application of Weil descent", *Codes and Cryptography*, Lecture Notes in Computer Science, **1746** (1999), Springer-Verlag, 191-200.

[15] R. Gallant, R. Lambert and S. Vanstone, "Improving the parallelized Pollard lambda search on binary anomalous curves", to appear in *Mathematics of Computation*.

[16] S. Garfinkel, *PGP: Pretty Good Privacy*, O'Reilly & Associates, 1995.

[17] P. Gaudry, F. Hess and N. Smart, "Constructive and destructive facets of Weil descent on elliptic curves", preprint, January 2000. Available from http://www.hpl.hp.com/techreports/2000/HPL-2000-10.html

[18] GNU Privacy Guard, http://www.gnupg.org

[19] I. Goldberg, "Pilot stuff from the ISAAC Group", http://www.isaac.cs.berkeley.edu/pilot/

[20] P. Gutmann, "Cryptolib". http://www.cs.auckland.ac.nz/~pgut001/cryptlib

[21] P. Gutmann, "Software generation of practically strong random numbers", *Proceedings of the Seventh USENIX Security Symposium*, 1998, 243-257.

[22] T. Hasegawa, J. Nakajima and M. Matsui, "A practical implementation of elliptic curve cryptosystems over $GF(p)$ on a 16-bit microcomputer", *Proceedings of PKC '98*, Lecture Notes in Computer Science, **1431** (1998), 182-194.

[23] D. Hankerson, J. Lopez Hernandez and A. Menezes, "Software implementations of elliptic curve cryptography over fields of characteristic two", draft, 2000.

[24] IEEE P1363, "Standard specifications for public-key cryptography", ballot draft, 1999. Drafts available at http://grouper.ieee.org/groups/1363

[25] The International PGP Home Page, http://www.pgpi.org

[26] D. Johnson and A. Menezes, "The elliptic curve digital signature algorithm (ECDSA)", Technical report CORR-34, Dept. of C&O, University of Waterloo, 1999. Available from http://www.cacr.math.uwaterloo.ca

[27] N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of Computation*, **48** (1987), 203-209.

[28] N. Koblitz, "CM-curves with good cryptographic properties", *Advances in Cryptology – Crypto '91*, Lecture Notes in Computer Science, **576** (1992), Springer-Verlag, 279-287.

[29] N. Koblitz, *A Course in Number Theory and Cryptography*, 2nd edition, Springer-Verlag, 1994.

[30] H. Krawczyk, M. Bellare and R. Canetti, "HMAC: Keyed-hashing for message authentication", Internet RFC 2104, February 1997.

[31] A. Lenstra and E. Verheul, "Selecting cryptographic key sizes", *Proceedings of PKC 2000*, Lecture Notes in Computer Science, **1751** (2000), Springer-Verlag, 446-465.

[32] A. Menezes, T. Okamoto and S. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field", *IEEE Transactions on Information Theory*, **39** (1993), 1639-1646.

[33] V. Miller, "Uses of elliptic curves in cryptography", *Advances in Cryptology – CRYPTO '85*, Lecture Notes in Computer Science, **218** (1986), Springer-Verlag, 417-426.

[34] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS Publication 186-2, February 2000. Available at http://csrc.nist.gov/fips

[35] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS Publication 180-1, April 1995. Available at http://csrc.nist.gov/fips

[36] P. van Oorschot and M. Wiener, "Parallel collision search with cryptanalytic applications", *Journal of Cryptology*, **12** (1999), 1-28.

[37] OpenSSL, http://www.openssl.org

[38] PGP versions, http://www.paranoia.com/~vax/pgp_versions.html

[39] David Poguet, *PalmPilot: The Ultimate Guide*, 2nd edition, O'Reilly & Associates, 1999.

[40] B. Ramsdell, "S/MIME version 3 message specification", Internet RFC 2633, June 1999.

[41] RIM Software Developer's Kit (SDK), http://developers.rim.net/handhelds/sdk

[42] R. Schroeppel, H. Orman, S. O'Malley and O. Spatscheck, "Fast key exchange with elliptic curve systems", *Advances in Cryptology – Crypto '95*, Lecture Notes in Computer Science, **963** (1995), Springer-Verlag, 43-56.

[43] I. Semaev, "Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p", *Mathematics of Computation*, **67** (1998), 353-356.

[44] J. Solinas, "An improved algorithm for arithmetic on a family of elliptic curves", *Advances in Cryptology – Crypto '97*, Lecture Notes in Computer Science, **1294** (1997), Springer-Verlag, 357-371.

[45] J. Solinas, "Improved algorithms for arithmetic on anomalous binary curves", Technical report CORR-25, Dept. of C&O, University of Waterloo, 1999. Available from http://www.cacr.math.uwaterloo.ca

[46] W@P white paper, 1999, http://www.wapforum.org/what/whitepapers.htm

[47] A. Whitten and J. Tygar, "Why Johnny can't encrypt: A usability evaluation of PGP 5.0", *Proceedings of the Eighth USENIX Security Symposium*, 1999.

[48] M. Wiener and R. Zuccherato, "Faster attacks on elliptic curve cryptosystems", *Selected Areas in Cryptography*, Lecture Notes in Computer Science, **1556** (1999), Springer-Verlag, 190-200.

[49] E. De Win, S. Mister, B. Preneel and M. Wiener, "On the performance of signature schemes based on elliptic curves", *Algorithmic Number Theory, Proceedings Third Intern. Symp., ANTS-III*, Lecture Notes in Computer Science, **1423** (1998), Springer-Verlag, 252-266.

[50] Wireless Application Protocol Forum, "Wireless Application Protocol", John Wiley & Sons, Inc., 1999. See also http://www.wapforum.org/

[51] Wireless Application Protocol Forum, "Wireless Transport Layer Security Specification", chapter 16 of [50], 1999. Drafts available at http://www.wapforum.org

[52] P. Zimmermann, *The Official PGP User's Guide*, MIT Press, 1995.

Shibboleth: Private Mailing List Manager

Matt Curtin
Interhack Corporation
http://www.interhack.net/people/cmcurtin/

June 15, 2000

Abstract

We describe *Shibboleth*, a program to manage private Internet mailing lists. Differing from other mailing list managers, *Shibboleth* manages lists or groups of lists that are closed, or have membership by invitation only. So instead of focusing on automating the processes of subscribing and unsubscribing readers, we include features like SMTP forgery detection, prevention of outsiders' ability to harvest usable email addresses from mailing list archives, and support for cryptographic strength user authentication and nonrepudiation.

1 Introduction

> Then Jephthah gathered together all the men of Gilead, and fought with Ephraim: and the men of Gilead smote Ephraim, because they said, Ye Gileadites are fugitives of Ephraim among the Ephraimites, and among the Manassites. And the Gileadites took the passages of Jordan before the Ephraimites: and it was so, that when those Ephraimites which were escaped said, Let me go over; that the men of Gilead said unto him, Art thou an Ephraimite? If he said, Nay; Then said they unto him, Say now Shibboleth: and he said Sibboleth: for he could not frame to pronounce it right. Then they took him, and slew him at the passages of Jordan: and there fell at that time of the Ephraimites forty and two thousand.
>
> Judges 12:4–6, KJ

Shibboleth was conceived in early 1995 as a system that would allow a group of people to communicate freely with one another without concern about outsiders being able to infiltrate the group or to address the group by impersonating one of its members. At the time, widely available mailing list software had no means of effectively addressing these security and privacy requirements.

Since that time, the Internet has seen explosive growth, which has unfortunately increased significantly the number of unscrupulous and greedy marketeers among us online. These have devised many avenues of finding people, arguably invading their privacy "to market to them more effectively", and engaging in abusive practices like spamming [16, 10]. On today's Internet, it is difficult to participate in a forum of any type—even "private" ones—without being exposed to the risk of making oneself known to the outside world and having one's address published for all to see and to abuse.

Shibboleth has managed to avoid falling victim to the abusive behavior that is sadly becoming increasingly popular on the Internet today.

More information about the system, including its source code, is available on the Web at `http://www.interhack.net/projects/shibboleth/`.

1.1 Terminology

It will be easy to get lost in these interactions unless we explicitly state exactly what we mean by the these terms.

Family A group of mailing lists on the same machine, managed by the same installation of *Shibboleth*.

List A specific mailing list.

Moderator Human who approves messages and manages elements of a list. Each list can have a single moderator or a set of moderators. A moderator can be a moderator for several lists.

Administrator A human who deals with irregularities, such indications of mail forgery and digital signature failures. Additionally, administrators are responsible for the management of the *Shibboleth* installation as a whole, including such things as global configuration options.

Hosts A group responsible for the operation of the family of lists. Typically, this is the moderators and administrators as a single group.

Outsider A network user who has no association with the family of lists in the particular *Shibboleth* installation.

Insider A user who does have association with this particular family of lists; someone with a profile in the members database.

Subscriber A particular type of insider: one who is subscribed to a particular list.

Nym The name by which an insider is known. This might be some construction of the insider's name (like "first_last"), or it could be a handle by which he is known.

1.2 Differentiating Ourselves

It is important to note that *Shibboleth* is vastly different from most mailing list managers because most are designed to take care of routine issues of subscriptions; our software makes no attempt to automate this process to the same degree. Subscriptions are by invitation only. That is, the hosts initiate the subscription process by sending an invitation. If the user accepts, a profile is created, and he is welcomed to the family of lists.

2 High-Level Design

Before construction began, we wrote about the system's design goals and requirements.

2.1 List Structure

Shibboleth thinks of lists in groups, which we call families. If a group of security folks wants to work together, it can do so by defining a "family", which might be called "White Hats". When we refer to a White Hats member, we mean only that *Shibboleth* has a profile for that user in its member database. That name is typically the basis of deciding what prefix to use to reference the family of lists as a whole. In this example, we'll use "WH".

Within each family are any number of lists that belong to that family. The only lists that *Shibboleth* expects to find by default are

- an "-all" list (which includes everyone in the database) and
- a list for the list's hosts, those responsible for the operation of the lists.

There is nothing inherently special about either of these lists. Any insider may submit a message to the "-all" list; whether these require moderator approval or are allowed at all is purely a matter of configuration, as is true with any other list.

Any number of other lists can be created. Their names consist of the prefix ("WH" in our example) and its separator ("-" in our example) followed by a keyword to identify the list. A WH list to discuss projects might be "wh-projects", and another to handle otherwise off-topic traffic might be "wh-chat".

Each list has its own privacy level. That is, some lists can be available for anyone associated with that list's family. Others can require approval from a moderator. Thus, if there's a topic that would not be open for all WH members, it can be discussed on a list marked "private". Members may only retrieve archival postings from non-private lists, or private lists to which they are currently subscribed.

In no case is mailing to a combination of insiders and

outsiders supported. *Shibboleth* will trigger an error for such messages, requiring administrator approval.

2.2 Design Goals

High-level goals for the system include

Members-Only Access Only insiders can send mail through the relay to any of the family's lists or other insiders.

Resistant to Forgery The system should be resistant to SMTP [14] mail forgeries [19]. Some basic header checking should be done. Additionally, the system should be able to verify and to generate digital signatures in order to make the possibility of convincing forgeries computationally infeasible.

Configurable Access Each member of the list will have an "access level" associated with his account. These are

Admin List owner; can do anything.

User Regular list member. Can read and post messages without special restrictions.

Novice List member, but read-only. (Novices may submit articles, but they must be approved by a moderator, irrespective of the list's configuration.)

Timeliness Users should not have to wait "long periods of time" for processing of their mail.

Minimal Overhead The administrative burden must be reasonably manageable.

- Day-to-day tasks pursuant to the operation of the mailing list should be minimal. It shouldn't require a lot of time to manage a mailing list.
- Processing of the list should not be so expensive that it bogs down the machine running the system.
- A simple-as-possible configuration file should exist which would allow configuration changes to be made easily, either manually or by some mechanism in the software itself.

Usefulness The system needs to provide general utility that would be expected from a mailing list package, including

- Ease of use: the learning curve for users should be gentle;
- Archival of old messages;
- Digest creation: send digests to those who want to receive only specified articles from the archive;
- File server: a means for files of interest to the user community to be sent via email.

2.3 System Requirements

Specific system requirements. Features of this section indicate the feature must be part of the original implementation. Features that can be added in a subsequent version of the software are listed in the next section.

2.3.1 Moderation

A list of moderators is assigned for each list managed by the software. Every time a message requiring moderator intervention is processed, one copy of the message is sent to each address in the list of moderators.

This design is suboptimal. We have found that it can work in cases where there are few moderators and they have some agreement whereby they can decide who will process which messages. Nevertheless, this is relatively cumbersome, and would best be replaced by a mechanism to allow a moderator to fetch a number of messages in the queue, or to inquire as to the number of messages in the queue.

Each list can be configured for one of several moderation modes.

Unmoderated Moderate nothing: let all messages pass;

Moderate new threads Require moderator approval for only the first message in any thread;

Taboos Moderate messages having a header matching a given pattern;

Unproven Require moderator approval, except for messages whose authors have been "proven" using an authentication mechanism;

Fully Moderated Require moderator approval for everything.

"Taboos" is actually a special case: one can, for example, employ both "moderate new threads" and "taboos". If any taboo patterns are specified, they'll be used. Any `Subjects` matching one of the taboo patterns will trigger the moderation rule, irrespective of any other moderation configurations for that list.

2.3.2 Sender verification

Each member of the list has a list of patterns used to identify his known addresses. When a message arrives, the `From` header is compared to patterns in the profiles in the database so that the user who sent the message can be identified.

If a message comes from an unknown address, it can be spooled for a moderator to approve or to reject the message. Additionally, in order to prevent an outsider who mailed an insider from getting the idea that the address he used is valid, a "user unknown" bounce message is sent.

As part of the verification process, the `Received` and `Message-Id` SMTP headers [6] are examined to decrease the possibility of forging a message that appears to come from a known (legitimate) user. If a forgery is suspected, the system spools the message for an administrator to peruse. In practice, this rule is most often triggered by administrative changes in the user's Internet Service Provider (ISP), such as the addition of previously unknown mail relays, or changes in the user's behavior, such as the use of a new ISP for IP connectivity without changes in the user's email address. (As an example, someone might have an "address for life" from a university and always use that. As far as anyone who sees only `From` and `Reply-To` headers is concerned, there is no change when such a person switches ISPs. However, someone looking at `Received` headers will be able to identify that mail is definitely coming from a different source when such a person changes ISPs.) The administrator simply replies to mail *Shibboleth* sent, updating the profile to include the new relay, or a new pattern that will cover the relay.

The optional `X-Password` field is used as an additional means of convincing the system of the message's authenticity. Thus, if the `X-Password`'s value matches the user's password in his profile, SMTP header errors are ignored.

Note that remote MTAs, like any other users, are outsiders. That means that if Alice, an insider, sends mail to Bob, another insider, via *Shibboleth* and Bob's MTA sends any kind of a message to Alice (perhaps to report a delay in delivery, a bounce, or some other error condition), Alice will never see the bounce. Such a message would have a `From` header like `postmaster` or `mailer-daemon` at Bob's site, which will not be in the members database. The bounce message that Bob's MTA tried to send will therefore generate a "user unknown" error condition for the administrator's attention and generate a "user unknown" bounce. This prevents attacks against the system where someone would be able to send a message to a *Shibboleth* user by generating a false bounce.

2.3.3 Address Standardization or Shadowing

Each user should have a standardized address, in the form of "prefix-nym". Some might want their nyms to be their first and last name. Others might like their nyms to be some sort of unique token. The prefix should make it clear that the intended target is a *Shibboleth* user. A private installation called "white hats" might have a prefix of "wh", thus if Alice is a member of "white hats", her address to other "white hats" members would be `wh-alice@example.com`.

This serves two purposes.

- Insiders can easily mail each other by knowing only the first and last names (or the nym) of their addressee.
- All mail sent this way is subject to the same defenses as mail sent to a list is. Hence, snoops seeing mail in transit from the list to its recipients will not be able to gather email addresses (which might later be used to target for mailings or for some other purpose unrelated to the specific message from which it was harvested). For this to work effectively, it is necessary to ensure that headers from the original message are not passed through the system, especially `Received`, `Message-Id`, `X-*`, and `Reply-To`.

It should be noted that mail from an insider to the

system can still be snooped. The effects of gathering addresses this way is far smaller than from the system to the list members, simply because the number of users affected is much smaller. The only effective way to gather all of the addresses of the system's users is to snoop all of the incoming traffic, which can only be effectively done on a point in the network that the system will exclusively use for its network connection.

2.3.4 Header Canonicalization

All mail from the system has a consistent `From` header format, which easily identifies the kind of mail that's been sent to the user by *Shibboleth*. For example:

.-. Alice mail from Alice to a *Shibboleth* mailing list.

.^. Alice Mail from Alice to you via *Shibboleth*.

.#. Alice Mail from Alice to the mailing list for list managers.

.!. List Managers Mail from the List Managers via the "-all" list (the broadcast channel).

This format makes it possible to score articles easily in software in addition to identify visually why the article was received. ("Scoring" is a method of having the user agent mark messages, either as likely to be of interest to the user or likely not to be of interest. Scoring is described in various FAQs on Usenet and Usenet client software [1, 11, 18].)

2.3.5 Archives

Insider-only retrieval of messages posted to a given list. Further, although "public" lists—those available to any insider—will be retrievable by all insiders, only subscribers of a private list can retrieve messages from that list's archive.

2.3.6 Portability

Should run unmodified on any Unix machine where Perl 5.004 can be found. In theory, it would take very little work for the system to run under alien systems like Windows NT and MacOS, provided that the local Mail Transfer Agent (MTA) has some means of running a process as a given user.

2.3.7 Logging

The complete and unmodified headers of each message are logged. This was implemented to aid in determining precisely which headers triggered particular rules and whatnot after the fact. At runtime, if an error is reported, the headers causing the error are included, but we thought there might be some value in being able to examine the headers after the fact, or to be able to examine the data for patterns over a period of time. This is also consistent with the belief that where security is a concern, it is better not to need what one has than not to have what one needs.

2.3.8 White Pages

This feature allows each user to have a bit of text that would be intended as a biographical sketch available to other insiders. Even among insiders, individual users have control over who sees their white pages entries, by providing their preferred means of handling requests to see their white pages entries. Available options are:

Deny Do not let anyone (but administrators) retrieve the entry.

Ask Respond to each attempt to see the white pages entry by mailing the entry's owner and asking for approval. If the owner approves the request (by replying to the request with a "yes"), forward the entry to the requester. Refusals (made by replying to the request with a "no") result in the requester being told that the request has been denied by the entry's owner. If no answer is received in a configurable amount of time, the requester is told that the request has expired without a response from the entry's owner.

Allow Let anyone who is a subscriber of the list view the entry.

2.3.9 Remote Administration

While the software should be able to be administered directly from the machine where it's running, a set of commands should be available to the administrators, to allow them to administer the list without having to login to the machine running the software. Additionally, if there are changes that need to be made on more than one machine, the software should accept the command, and then sync the other systems to ensure that all systems are configured to agree with each other.

Because of the nature of Internet email and the potential effects of accepting commands from attackers, the software does not execute commands unless it can be proven that they originated from an administrator. For this purpose, a valid PGP digital signature is required. Recently, we have added the ability for administrators and moderators to authenticate themselves via the use of the `X-Password` header mechanism. We don't recommend use of this mechanism; if it's at all possible, use PGP.

2.3.10 File serving

Having files (such as system documentation, or other files of interest) available for insiders can be useful. Making them available via HTTP [7, 3] or FTP [15] could be problematic, as these are not subject to the same sorts of authentication mechanisms that exist in *Shibboleth*. Making *Shibboleth* do this itself has been most useful; requests are sent to *Shibboleth* as are any other requests, and responses are sent via email, with the requested files sent as MIME [8] attachments.

2.3.11 PGP Signature Generation

Each list has the option of having all of its traffic PGP signed [2]. That is, before *Shibboleth* sends a message, it PGP signs the message with its own key. This is an important option: all of the header checking in the world won't do us any good if a user can be fooled by a simple forgery that never went through the *Shibboleth* server. By enabling the option to have all messages PGP signed, such forgeries wouldn't be possible to perpetrate in a way that would fool most of the users. They'd want to know where the signature is, or why the signature doesn't verify properly.

2.3.12 Additional Sender Verification

In addition to SMTP header checks and the use of the `X-Password` header, *Shibboleth* supports PGP. A message signed with PGP will be verified on the basis of the signature's correctness. SMTP header checks will not be performed.

2.4 Additional Features

These are features that we should have, but have not yet implemented.

2.4.1 Peer support

Because of the processing requirements of the system, it is desirable to have the ability to share the load among several machines. Perhaps one machine receives incoming mail and handles verification, while another machine receives the mail from the first machine and handles subsequent processing. Once a more final version of the features are available, we'll describe each of the parts' interfaces, which will show where load can be split among machines.

2.4.2 Configurable Load

The software should be configurable to be a system pig (i.e., handle redundant tasks with parallel processes or threads) or to be nice (i.e., process everything sequentially; do not take up extra cycles.)

2.4.3 Digest Generation

Shibboleth's digest is different from most systems' digests. A digest can be requested by any user and an index of all of the articles which have been posted to the specified list will be returned. The user then chooses which articles he wants by quoting the lines containing the articles he wants to read and sending that mail back to the digester. The digester then sends those articles to the requester, separately.

Something we're considering is the ability for users to receive mail in batches, perhaps through some sort of digest capability as described in RFC 1153 [21].

3 Implementation

We'll limit our focus on implementation details that we believe to be the most relevant to our goals of privacy and security, particularly where we do things that aren't known to be done by other mailing list managers.

3.1 Language

Perl was chosen as the implementation language because it satisfied some important criteria for our application.

Portability Well-written Perl code will run unchanged on essentially any Unix implementation, and even on non-Unix platforms.

Safety Perl frees the programmer from dealing with the sorts of problems that are unrelated to the project at hand and historically the most problematic, including management of memory and of fixed-length buffers. Additionally, Perl's taint-checking allows us to run *Shibboleth* without worrying about the likelihood of unverified user data being handled unsafely.

Rich Library Perl has a rich library of modules that allows us to work with simple interfaces to protocols and systems we'll be using.

Pattern Matching Perl's sophisticated pattern-matching capabilities are well suited to the sort of analysis of text data that we do.

3.2 Sender Verification

Erring on the side of paranoia, we begin with the assumption that a message is from an outsider. We'll consider the verification process first and then consider the noteworthy parts in more detail.

1. If a digital signature is present it is evaluated.
   (a) If the digital signature is good, the sender's identity is accepted without need for further verification.
   (b) If the digital signature is bad, the message is sent to the administrator for review, with the note that the digital signature failed.
2. If the user can be identified, his nym is associated with the message. Otherwise, the message is sent to the administrator for review with the note that the sender could not be identified as an insider and a "user unknown" bounce is returned to the sender.
3. The `Received` and `Message-ID` headers in the message are examined. If any hosts that are not in the user's profile were involved in sending the message, the message is sent to the administrator for review, with the note that a header didn't match the user's profile.
4. If the user's profile contains a password, the message is checked for the presence and correctness of an `X-Password` header.

3.3 Getting Mail to *Shibboleth*

Mailing lists typically require a few aliases to ensure that mail directed to the list's address will be delivered properly, as well as such variations as `listname-owner`. Because we need *Shibboleth* to process not only things directed specifically to it or to any of its lists, but also to any other insider, we would require a larger number of aliases.

Shibboleth has the support necessary to generate and to maintain a separate MTA alias file.

We actually favor a newfangled option[1] to all of this alias maintenance. Most MTAs support a feature for aliasing multiple addresses to a single mailbox. For example, in recent versions of *Sendmail,* mail addressed to `user`, `user+foo`, and `user+bar` all get delivered to the same place. A Usenet FAQ describes how to do this in detail [13]. By using this feature, a site can create a user with a prefix that will be used for all of the family's lists. Then, addresses will be `user+list` or `user+nym`. *Shibboleth* will generate bounces for anything that doesn't exist. That is, if "nym" does not exist, it is be

[1] This options is not fully implemented.

Shibboleth's responsibility to inform the sender of the incoming message that the intended recipient address does not exist.

By having mail go to a user account, we can also place the burden of managing the ID of the user running the process on the MTA. Rather than having to write our own `setuid` front-end for *Shibboleth*, the MTA will perform the `setuid` for us. This saves us some headaches in permissions related to aliases and other files we need to access. It's also much more easy for us to run from a single unprivileged account this way, making all of our database files, archives, and other data unreadable to all other system users.

3.3.1 Identifying the User

Each user's profile, as is shown in Figure 1 contains a list of Perl patterns; an address that matches the pattern is associated with the user. This allows users to send mail from any of their accounts, without revealing how many such accounts they have, or even giving anyone any idea that they have more than one. This is a convenience issue, as mailing lists that accept mail only from subscribers are typically less intelligent and require that one send mail from the exact address that one uses to subscribe to the list. For those who do a lot of contract work (thus changing daytime address often), those who send mail from both home and work, or those who use more than one machine, our solution is much more workable.

Headers that we examine for this case are the SMTP envelope's `From` (sometimes called `From_` because MTAs have historically stored the value in a `From` header without a separating colon, but instead followed by a space and a timestamp) and the message's `From` header.

3.3.2 SMTP Header Checking

In an effort to prevent outsiders from being able to send mail by impersonating an insider and to prevent insiders from impersonating each other by means of trivial SMTP forgery, we examine each of the headers put in place by mail relays used to deliver the message as well as the "domain part" (right side: that which follows `@`) of the `Message-ID`.

```
wh-matt_curtin is Matt Curtin

Destination Email: cmcurtin@interhack.net
Valid addresses:   cmcurtin@interhack\.net
            cmcurtin@\w+\.interhack\.net
            cmcurtin@cis\.ohio-state\.edu
MX servers:  \w+\.interhack\.net
            \w+\.cis\.ohio-state\.edu
X-Password:
White page access:     allow
PGP key ID:            DEADBEEF
User is:
User subscribed:       wh-hosts, wh-all,
           wh-chat, wh-security
User is admin in:      wh-all
User is moderator in: wh-all
```

Figure 1: Typical User Profile

Again, each insider's profile contains a list of Perl patterns used to identify known SMTP relays. If any relay does not match one of the patterns in the profile, an error is signaled.

For convenience, administrators can specify "clusters", a token that will be associated with a list of patterns. So, for example, if an installation has some number of insiders who are all AOL subscribers, each can have the `*aol*` cluster in his profile, and then the `*aol*` cluster can be made to recognize hosts that fit the pattern `\S+\.(mx|mail)\.aol\.com` and `mrin[0-9]+`.

3.3.3 X-Password

We recognize that PGP—our preference for authentication—is not available to all. Nontechnical members on less capable platforms might especially have difficulty using PGP correctly and finding tools that allow them to use it conveniently. Lastly and ironically, there are companies whose security organizations have banned the use of PGP.

To provide an option for additional authentication for those who have no capability to PGP, *Shibboleth* supports a user-defined header that contains a password. This isn't considered a "high-security" option, as it's susceptible to replay attacks [9], just as is any reusable-password authentication scheme whose credentials are sent in cleartext.

It is also important to note that use of PGP also provides the feature of nonrepudiation. If the `X-Password` feature is used instead of PGP, there is no nonrepudiation feature in the system.

3.4 Address Standardization

Several benefits are realized by the address standardization feature. A common problem with mailing lists is that responses to messages sent to mailing lists will have both the mailing list itself and the author of the message that prompted a response included on the copy-to list, resulting in some subscribers receiving several copies of messages in response to theirs or of messages even further down the thread.

Implementation of the address standardization mechanism requires that all messages—both insider-to-insider and insider-to-list—be processed by *Shibboleth*. As such, we can prevent subscribers from receiving multiple copies of the same message, even if the author specifies a specific user's address and a list to which he subscribes.

Sometimes, a *Shibboleth* user's shadowed address will fall into the hands of outsiders, either by way of oversight, perhaps a careless person forwarding the mail to outsiders without removing such headers, or intentionally. Perhaps a subscriber has been ejected for some reason and has saved addresses of other insiders.

An outsider sending a message to an insider or addressing one of the *Shibboleth* lists will receive a "user unknown" bounce. A copy of the message will also be sent to the list administrators. This has proven to be an effective means of preventing unwanted traffic from finding its way to insiders. Never has unsolicited bulk email ("spam") ever made it to a subscriber, though some of our addresses appear to have been sold as part of at least one spam software package; the administrators got copies of the spam, as they would any message from an apparent outsider, but the insiders had no idea that the spam was ever directed their way. Individuals who unscrupulously add others to their lists without confirmation occasionally add an insider's shadowed address. After receiving the bounce, the list operator who added the shadowed address will typically remove the address. (If not, he will just keep getting bounces!)

If an insider sends mail that includes outsiders, the message goes to an administrator for handling. If the administrator approves the message, *Shibboleth* will remove the outsiders' addresses, thus preventing any replies from insiders also being directed to outsiders. Replies from outsiders that include insider addresses will, of course, bounce.

3.5 Cryptographic Strength Moderation

A potential weakness for any moderation scheme is the authentication mechanism used for the moderators to identify themselves and the messages that they approve. As *Shibboleth* is fully integrated with PGP (albeit "classic" IDEA/RSA/MD5 PGP 2.6.x), we can easily require cryptographic strength moderation. Thus, defeating the moderation scheme would require a crack of a moderator's key, or the ability to forge an MD5 hashed signature, both of which are currently computationally infeasible.

Administrative functions also require the same level of authentication.[2]

Messages to be moderated appear to be bounces to most mail user agents (MUA). MUAs that support a feature like "retry bounce", such as Kyle Jones's excellent *VM* for XEmacs and GNU Emacs, will work best, as the submissions can be read from the moderation queue, brought into a composition buffer, PGP signed, and then sent on their way.

Shibboleth, upon seeing that the message has a valid PGP signature belonging to a moderator, will send the message on its way. It is noteworthy that only the part of the message that was signed will be passed. Anything outside of the delimiters for the signed message will not be included; neither will the PGP signature itself.

Because we use a version of PGP that is not MIME-aware, PGP does not interfere with any of the MIMEisms that are in the original message. Its `Content-type` header is left unchanged, and the signature engulfs the entire message body, including all attachment data. Thus, when *Shibboleth* verifies the

[2] Again, we recently added the ability to allow the simple `X-Password` header authentication for moderators and administrators. This isn't something we recommend that administrators enable, but provide it as a means of authentication where PGP isn't an option.

signature, it is verifying not only the content of the message, but also of all of the attachments. Any changes to the attachments in transit (such as, for example, the addition of a virus-infected file) would cause the signature to fail, causing the attempted approval message to go to the administrator for review.

3.6 Load Sharing

Instead of requiring that all processing and all outbound messages be sent via the same host, *Shibboleth* provides the ability to direct its outbound mail to another host. This allows the system running *Shibboleth* to share the burden of running the list: one machine can receive all of the incoming messages and handle that processing; another can be used to send all of the outbound messages.

Other features that would enable a more granular level of load sharing have been planned.

3.7 Outgoing Messages

To prevent the leakage of header data that could provide information about a poster's address, *Shibboleth* does not simply forward the message to the appropriate users. *Shibboleth* actually creates a new message altogether. *Shibboleth's* configuration contains a list of Perl patterns whose headers should be preserved. This allows us to include things like `Subject`, without having to worry about what data might bleed in other headers.

Outgoing messages have their own `Message-ID` headers generated, thus preventing the poster's MTA's domain name from being present in outgoing messages. Thus, all `Message-ID`s have the site name of the MTA that *Shibboleth* uses. As such, it is safe to configure *Shibboleth* to pass headers like `In-Reply-To` and `References`, which can be used for building threads. These headers should contain only `Message-ID`s that were generated by *Shibboleth*.

In practice, we also allow such personal touches that are contained in headers like `X-Face` and `X-Attribution`. MIME works by passing such headers as `MIME-Version` and `Content-type`. This is all completely under the control of the list family administrator.

For the sake of safety, *Shibboleth* adds an `X-Loop` header and will recognize its own token as a means of preventing mail loops. If the header and token are present in an incoming message, *Shibboleth* will not deliver the message; it will exit, logging the detection of a mail loop.

4 Administration

Administration of a *Shibboleth* installation tends to be somewhat intensive at first, while the system learns different addresses the insiders will use and what mail relays will deliver their mail. Once the system has been running for a while, the regular posters will have their profiles set properly, and the system will not report problems for them except in the cases of real errors or in major system configuration changes (such as an ISP buying another and changing an insider's address).

The additional administration comes from the need to train the system to recognize the legitimate mail relays that are used by various insiders. Because we use Perl patterns, one need not list every possible relay host for each user: to allow any host with an alphanumeric name inside of the zone `example.com`, we can specify the pattern `\w+\.example\.com`.

Beyond this, there is little difference in the administrative operation of a *Shibboleth* list, as compared to another list manager, such as *Majordomo* [5]. Because of our authentication system's requirement for strong authentication, it isn't practical for us to implement a web-based interface like that of *Mailman* [20].

5 Future Work

We have identified some areas where *Shibboleth* could be improved.

5.1 Other Secure Mail Standards

Support for emerging standards, such as OpenPGP [4] (which supports newer, sometimes more desirable options for ciphers and

hashing algorithms) and perhaps S/MIME [17], would be useful. There are some tricky matters to be resolved here, especially in allowing a moderator to sign a signed message and sanely preserving MIME `Content-type` data, but it seems well within reach.

An additional benefit to support of these standards would be that one could reasonably sign all outgoing messages without the need to be concerned whether such signing would create problems for MIME-formatted messages. (Our RFC 1991-compliant PGP, though convenient for approving MIME-formatted messages, isn't practical for signing things that will be reviewed by most MUAs. Our option to have *Shibboleth* sign all outgoing messages would prevent most MUAs from being able to decode MIME-formatted data properly. If we could perform the signing in a way that's friendly with MIME, the sign-outgoing-mail feature would be much more useful.)

5.2 Better Peer Support

Especially in an installation where many users digitally sign their submissions to the mailing list and where the system is signing each message it relays, CPU overhead could be significant. It would be nice if there were a more intelligent way to spread the load among some set of hosts, rather than placing all of the processing burden on one host and only providing the option of giving the delivery burden to another.

5.3 Better Moderation Scheme

Currently, there is a race condition for lists where there are multiple moderators: all moderators for a particular list get all copies of messages for that list to be moderated. As a result, multiple approval (and thus, multiple posts) are possible. We have managed this problem by convention, but a more intelligent system for moderation would include the ability for a moderator to get some number of messages from the moderation queue and then work on them himself, without leaving the possibility of duplicating another moderator's work.

5.4 Tolerance of SMTP Irregularities

Our current implementation treats any irregularity as an error that could indicate forgery, thus requiring administrative attention. We have been thinking about ways of lightening this load somewhat without losing the benefits of SMTP header verification.

The most interesting solution to this problem we've considered so far is the addition of a runtime parameter that will specify the number of irregularities that can be tolerated per message. Another runtime parameter could be used to indicate whether *Shibboleth* should automatically add the previously unknown relays to the user's profile if they are fewer than the specified number. Messages that have come through more unknown relays than that number could continue to be handled in the manner that today's errors are: administrative attention. Indeed, if they are not, messages will either mysteriously "disappear", or we'll override any benefits derived from checking the headers at all.

5.5 Local User Dilemma

People with local access via normal user accounts to systems that provide network services can often create problems for those network services. *Shibboleth* is no exception. Specifically, users who can inject a message locally can fool our SMTP header verification. Today, we do a simple enumeration of relays and compare those hosts to patterns in a user's profile. If any pattern in a user's profile does not match, no error is signaled. This is a mildly difficult problem, since the primary use for these lists of patterns is the allowance of posters to send either from home or from work. Were we to trigger an error if a pattern does not match, a message from such a user would need to be sent from home *and* work at the same time.

The best solution to this problem is to improve the intelligence of the SMTP header parsing. Rather than being a simple enumeration of hosts to be checked against a pattern, the headers should be parsed, thus telling us which host injected the message into the network and which hosts merely relayed the message.

Even in such a scenario, we're at the mercy of MTAs

that are out of our control. Perhaps in practice, this will be good enough to provide us much more security than we have now. Whether this is true will probably depend on each installation, where its users originate their mail, and which MTAs those systems run.

Depending on the MTA and its configuration, local users also have the ability to determine if a "user unknown" bounce is legitimately generated by the MTA or if it has been tricked into returning such an error.

For the time being, we have to consider local users "trusted", and reducing such trust is an area for further study.

5.6 Reducing Necessary Trust in Administrators

As implemented, *Shibboleth* users must trust the administrators. Though insiders are protected from outsiders and from each other, they are not protected from administrators. Also, all administrators have full administrative access to the system.

We're particularly intrigued by the possibility of future releases of *Shibboleth* actually distrusting administrators and hiding sensitive information—such as the members database—from them. Rather than having any single administrator being able to query the database for full user profiles, for example, what if we were to require that multiple administrators' signatures would be necessary to have *Shibboleth* reveal such information or perform especially "risky" operations?

Another interesting possibility is where hosts would be used for initially introducing new insiders to the system, after which users would be individually responsible for maintaining their own profiles. (It seems obvious that this would work only for rather experienced or technical users, but could the system be made to be smart enough that this is no longer true?)

6 Conclusions

We have shown that it is possible for a group of people who wishes to keep to itself can do so, even in today's Internet. Despite the lack of strong authentication mechanisms for email at any level other than application, it is possible to identify mail from insiders, letting it flow normally, without requiring that mail from outsiders flow just as freely. Even less-than-perfect schemes, like SMTP's simple headers, can be employed to determine reasonably the authenticity of a message.

Forcing all mail to insiders, public and private, to run through *Shibboleth* solves the problem of posters receiving multiple copies of posts later in the threads to which they contribute.

Because only *Shibboleth* nyms are known, someone cannot infiltrate the group, collect messages, and then use those addresses for his own nefarious purposes later. Once someone is no longer an insider, those addresses immediately become useless to him.

A simple checklist of features and requirements will show that we have satisfied our requirements. Our experience with running mailing lists with *Shibboleth* and enduring attacks against it gives us evidence that these safeguards have in fact worked.

On a daily basis, we hear about the difficulty of privacy and security on the Internet. Though not a substitute for the needed support for security in our infrastructure through efforts like IPsec [12], making better use of what we have available can move us in the right direction and bring us much closer to the sort of well-behaved system that does our bidding, and only our bidding.

7 Acknowledgments

Much of the first implementation of the original design owes its existence to Eugene Sandulenko (Женя Сандуленко). Thanks to Ed Sheppard for suggesting the name and to Carolyne J. Butler for careful proofreading.

References

[1] Usenet software: History and sources. Usenet FAQ, February 1998. [online] http://www.faqs.org/faqs/usenet/software/part1/.

[2] D. Atkins, W. Stallings, and P. Zimmermann. PGP Message Exchange Formats. RFC 1991, August 1996.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, May 1996.

[4] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP Message Format. RFC 2440, November 1998.

[5] D. Brent Chapman. Majordomo: How I manage 17 mailing lists without answering "-request" mail. In *Systems Administration (LISA VI) Conference*, pages 135–143, Long Beach, CA, October 19-23 1992. USENIX.

[6] D. Crocker. Standard for the format of ARPA Internet text messages. RFC 822, August 1982.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[8] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045, November 1996.

[9] N. Haller and R. Atkinson. On internet authentication. RFC 1704, October 1994.

[10] S. Hambridge and A. Lunde. DON'T SPEW A Set of Guidelines for Mass Unsolicited Mailings and Postings (spam*). RFC 2635, June 1999.

[11] Peter J. Kappesser. trn newsreader FAQ, part 2: Advanced. Usenet FAQ, August 1998. [online] http://www.faqs.org/faqs/usenet/software/trn-faq/part2/.

[12] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[13] Eli Pogonatus. Email addressing FAQ (how to use user+box@host addresses). Usenet FAQ, December 1998. [online] http://www.faqs.org/faqs/mail/addressing/.

[14] J. Postel. Simple Mail Transfer Protocol. RFC 821, August 1982.

[15] J. Postel and J.K. Reynolds. File Transfer Protocol. RFC 959, October 1985.

[16] Jon Postel. On the Junk Mail Problem. RFC 706, November 1975.

[17] B. Ramsdell (Ed.). S/MIME Version 3 Message Specification. RFC 2633, June 1999.

[18] Justin Sheehy. Gnus (emacs newsreader) FAQ. Usenet FAQ, January 2000. [online] http://www.faqs.org/faqs/gnus-faq/.

[19] Bob Thomas. On the Problem of Signature Authentication for Network Mail. RFC 644, July 1974.

[20] John Viega, Barry Warsaw, and Ken Manheimer. Mailman: The GNU mailing list manager. In *Twelfth Systems Administration Conference (LISA '98)*, page 309, Boston, Massachusetts, December 6-11 1998. USENIX.

[21] F. Wancho. Message Digest Format. RFC 1153, April 1990.

ISBN 1-880446-18-9